UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

Departamento de Arquitectura de Computadores y Automática

TESIS DOCTORAL

Técnicas de planificación en entornos reconfigurables para aplicaciones multimedia

Scheduling techniques in reconfigurable environments for multimedia applications

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Juan Antonio Clemente Barreira

Directores:

Daniel Mozos Muñoz
Jesús Javier Resano Ezcaray

Madrid, 2011

ISBN : 978-84-694-8487-6

UNIVERSIDAD COMPLUTENSE DE MADRID

Facultad de Informática

Departamento de Arquitectura de Computadores y Automática

Técnicas de Planificación en Entornos Reconfigurables para Aplicaciones Multimedia

Tesis Doctoral

Juan Antonio Clemente Barreira

Madrid, 2011

UNIVERSIDAD COMPLUTENSE DE MADRID

Facultad de Informática

Departamento de Arquitectura de Computadores y Automática

Técnicas de Planificación en Entornos Reconfigurables para Aplicaciones Multimedia

Scheduling Techniques in Reconfigurable Environments for Multimedia Applications

Tesis Doctoral

Ph. D. Thesis

Juan Antonio Clemente Barreira

Madrid, 2011

Técnicas de Planificación en Entornos Reconfigurables para Aplicaciones Multimedia

Memoria presentada por Juan Antonio Clemente Barreira para optar al grado de Doctor con Mención Europea por la Universidad Complutense de Madrid, realizada bajo la dirección de D. Daniel Mozos Muñoz y D. Jesús Javier Resano Ezcaray.

Madrid, Marzo 2011

Scheduling Techniques in Reconfigurable Environments for Multimedia Applications

PhD dissertation presented by Juan Antonio Clemente Barreira in partial fulfilment of the requirements for the degree of Doctor of Philosophy with European Mention to the Universidad Complutense de Madrid, supervised by Mr. Daniel Mozos Muñoz and Mr. Jesús Javier Resano Ezcaray.

Madrid, March 2011

Este trabajo ha sido posible gracias a la financiación de la Comisión Interministerial de Ciencia y Tecnología, a través de los proyectos *TIN2009-09806*, *AYA2009-13300-C03-02* y *PR34/07-15821*.

This work has been supported by the Comisión Interministerial de Ciencia y Tecnología from the Spanish government under research grants *TIN2009-09806*, *AYA2009-13300-C03-02* and *PR34/07-15821*.

Acknowledgements

It all started at some point in December 2005. I had not even finished my first year in my Masters Degree and I had been granted a scholarship to do some research in collaboration with one of the departments of the Faculty of Computer Science where I was a student. There I was, not knowing exactly what a research work was about and looking for someone in the Computer Architecture Department to work with. Then I met **Daniel Mozos** and **Javier Resano**, with whom I started a successful collaboration which results I present now in this PhD dissertation. Hence my first acknowledgement must be for them, who have made an outstanding job guiding this doctoral thesis.

The second acknowledgment must go to **my parents**, who have always given me all their best support and fondness throughout these four years, even though they don't know what on Earth am I talking about when I recite them the title of my doctoral thesis by heart (I am saying this in an affectionate way). And to my brother **Raúl**, who is still wondering why I am staying at the University whereas he is so looking forward to

becoming an engineer and never getting in touch again with the academic world whatsoever.

To my aunt **Angelines**, for her continuous interest about how things are going, either when I stay at home or I travel abroad. From now on, you will have the pleasure to call me *dottore* whenever you want to. And to my lovely cousin **Arantxa**, who still cannot believe how I ended up working on this thing called PhD (sometimes I cannot believe it either). You have transmitted me your desire to live by teaching me that the best way to cope with a breast cancer is to grin and bear it. We still have a pending trip when you will overcome that decease. Because of course you will.

To my neighbor **Nachete**, my makeshift personal trainer, who helps me to unwind from the work routine through a daily training plan in the gym, and in order to stay in shape too. To the people from the English classes during these last three years, especially **Rebeca**, **Laura** and **María**. Because, after all, I know that you don't actually think I am a living English dictionary. To my former classmates from University and however, friends: **Adolfo**, **Álex**, **Pedrosa**, **Pablo**, **Carlos**… Even though we don't see each other lately (which is too bad), every time we guys meet is really special.

I also want to dedicate a few words to my good old friend from High School **Víctor**, who has been literally always there to remember me what really good friends are for. And to **Lorena**, for caring about me as if she was my actual sister. You guys are a case study of true friendship. Thank you for giving me so many good moments and memories and for offering me your best support in my bad moments.

Professionally and personally speaking, many other people have had a very positive influence on me during these last years. I especially have to thank **José Luis Vázquez**, **Guillermo Botella**, **David Sigüenza, Hortensia Mecha** and **Julio Septién** for their kind hospitality when I first came to work

in the Faculty as a teaching assistant. And of course, I have to mention the *former and current dwellers of the Office 347*. To **Pablo García** and **Miguel Peón** for their technical support throughout this doctoral thesis (you guys are the best gurus about Computer Science that a PhD student can ever have). And to **Alberto Del Barrio**, **Fran Rincón**, **Carlos González**, **Jesús Fernández**, **Ángel Luis González**, the newcomers **José Luis Martínez**, **José Luis Lucas** and **Íñigo San Aniceto**; as well as **David Cuesta**, **Antonio Artés** and **Javier Sánchez Jurado**. With all of you I have shared labs, exams, conferences, trips, meals, pub crawls on *pretended Fridays* (which were actually Wednesdays or Thursdays, who cares) and football matches, among many others. Please believe when I say that this has made my daily life much easier inside these four walls.

An important part of these acknowledges has to go abroad and travel to Lausanne, Switzerland. Nothing would have been the same had it not been for my internships in the École Polytechnique Féderale de Lausanne (EPFL). In this sense, I really have to thank **David Atienza** for kindly inviting me to work in the Embedded Systems Laboratory (ESL); **Vincenzo Rana**, for guiding my research work there; and **Homeira Salimi**, for helping me with so many complex administrative procedures. I also have to mention my main comrade-in-arms, **Ivan Beretta**, who always helped me whenever I was stuck in some work-related issue through one of those *blackboard and couch meetings* (with even a *label removal* in there). While writing these lines, I'm already missing them so much. It has been a great pleasure to work with all of you and I really hope that our current research collaborations will continue in the years to come.

I also have to thank **Emiliano Rincón** and **Abel Míguez** for their priceless help for helping me to find a place where to live in Lausanne when I was just a newcomer, as well as for giving me wise advices when the only idea I had about the Helvetic country was through postcards and pictures. And

to my ex-flatmate **Albéric Magnard**, for helping me so much during that winter between 2009 and 2010 with everything in my daily life there, and for his frequent cheer-up mails while I was finishing this PhD dissertation.

A lot of people at ESL have had a very positive influence on me too: **Joaquín Recas**, **Hossein Mamaghanian**, **Alessandro Vincenzi**, **Michelangelo Carrozzo** (also known as ***Mr. Cappozzo***), **Andrea Cazzaniga (*Chezz*)**, **Rubén Braojos**, **Abdulkadir Akin**, **Karim Kanoun**, **Ahmed Dogan**, **Mohamed Sabry**, **Shivani Raghav**, **Arvind Shridar** and **Martino Ruggiero**, as well as the aforementioned **Emiliano**, **Abel**, **Pablo** and **Fran**. Interacting with so brilliant people has taught me a lot, even though I have been able to pick up just a few things. In addition, those out-of-lab moments such as barbecues by the Lake Geneva, raclettes, fondues, hikes (with a "*Ring of Fire*" as background), pubnights… demonstrate that ESL was actually like a little family. And I feel so lucky to be part of it.

And last but not least, I want to mention all that bunch of lovely people from all over the world that I had the great pleasure to meet in Switzerland and with whom I hope to keep friendship forever: **Juanfran**, **Alicia**, **Mar**, **Diego**, **Lucia**, **Álvaro**, **Alba**, **Jessica**, **Pepe**, **Pasha**, **Oriol**, **Yvonne**, **Susana**, **Gil**, **Leticia**, **Laura**, **Patricia, Patrizia**, **Jose**, **John**, **Marilyn**, **Dawn**… (please forgive me if I miss somebody important). I will never forget those ski weekends, hikes (with and without water at -200º C), alpinism sessions, trips, *splash splash*'es at the Lake Geneva, Jungle Speed games, bike rides, pubnights, Spanish-French and Spanish-English tandems, the World Cup 2010… I keep those moments very alive in my memory and I really hope to see you soon again. Because one thing is for sure: I will be back sooner or later.

Thank you all so much. A little piece of this PhD is also yours.

Contents

List of figures

List of tables

Quien la sigue, la consigue

Spanish proverb

Chapter 1: Introduction

Many current applications, such as signal/image processing, multimedia standards and vision applications have become more and more computationally intensive in the last few years. For this reason, embedded execution platforms have evolved into complex Systems-On-Chip (SoC) which, however, usually lack the flexibility needed to adapt the platform to different execution contexts in dynamic scenarios. In this sense, reconfigurable hardware (RHW) has appeared as a promising technology that offers an interesting trade-off between flexibility and performance, which many recent applications for embedded sytems demand.

The first chapter of this PhD dissertation presents an overview of RHW, which has been used as work environment for this research work. Thus, Section 1.1 presents this novel technology as a solution that combines both the flexibility of

the software and the high performance of a hardware design. Then, Section 1.2 presents different well-known challenges of the use of reconfigurable devices, focusing on those that are related to this research work. Finally, Section 1.3 introduces the multi-tasking system that I have used as work environment for this doctoral thesis.

1.1. Reconfigurable hardware

In May 1960, Gerald Estrin, a highly respected mathematician and physicist, considered nowadays as the father of modern reconfigurable computing, published an article in the Western Joint Computer Conference entitled "*Organization of Computer Systems – The Fixed Plus Variable Structure Computer*" [Estr60]. In this paper, he raised the concern of John Pasta, one of his colleagues at the University of California Los Angeles (UCLA), about many vital computational problems which solutions were beyond the capabilities of existing electronic computers. In his opinion, commercial computer manufacturers had lost interest in exploring risky and innovative computer architectures. Dr. Pasta challenged him to find new ways to organize computer systems in a way that it supposed a revolution in commercial computers at that time. As a result of this challenge, Dr. Estrin proposed a "fixed plus variable structure computer". It would consist of some "reconfigurable" hardware, controlled by a fixed general-purpose computer, which was intended to take advantage of both the high performance of the hardware and the flexibility of the software. Although Dr. Estrin built a demonstration machine, that idea didn't catch on mainly because of the lack of the available technology to implement such a system. In fact, in the meantime, microprocessors proved to be powerful enough to deal with increasingly complex applications in the years to come.

For many years, new applications and algorithms were being implemented in general purpose processors, which employed an instruction-stream-based von Neumann (vN) paradigm [HP95]. Thus, a software application was described as a sequence of instructions, which lead the operations to be performed step by step. However, as applications were becoming more and more computationally intensive, vN-based computers started to lack the required performance. Hence, additional support was required. A straightforward way to achieve this was to add hardware accelerators, which had been typically implemented as *Application Specific Integrated Circuits* (ASICs).

These kinds of circuits are customized for a particular use, rather than for general-purpose use. Hence, they are very likely to obtain the desired performance of any application, as well as low power consumption and small chip area. However, this comes at the price of adding high *Non-Recurring Engineering* (NRE) costs and the sacrifice of flexibility, since ASICs have very low area reusability. Thus, when these circuits become out-of-date, it is necessary to design a new one. This leads to new NRE costs and a high time-to-market, which is frequently a key factor for the success of a platform. In addition, as their functionality is fixed, ASICs cannot be updated in order to recover from detected bugs. For instance, this may be a major problem in space missions, since in the outer space an embedded system is exposed to external forces, such as particles coming from solar flares or cosmic radiation. These phenomena can lead to transient or permanent faults in the embedded system, and greatly increase the cost of the mission in terms of maintenance if the system is not able to recover from such errors.

For these reasons, in the 1980's, RHW underwent a renaissance that continues today. Both for research and commercial purposes, this innovative kind of computing has proved to be an efficient alternative to purely software-based and hardware-based solutions. From its birth to the current days, reconfigurable devices have experienced several evolutions in terms of performance and amount of reconfigurable resources, among others. However, the concept of reconfigurable computing has remained the same: to combine in a single device both the flexibility of a software program and the speed of a hardware circuit. Thus, RHW can be seen as something in the middle between these two ends, Figure 1.1 shows.

The main feature of RHW is the ability of altering the functionality of a hardware design by "loading" a new circuit on the reconfigurable fabric of the device. In addition, if the device is able to dynamically change its functionality at run time (i.e. without having to reboot the system), we talk about dynamically reconfigurable hardware (DRHW). Nowadays, most of the existing reconfigurable

devices are actually dynamically reconfigurable; hence in the following I will refer to DRHW rather than to RHW.

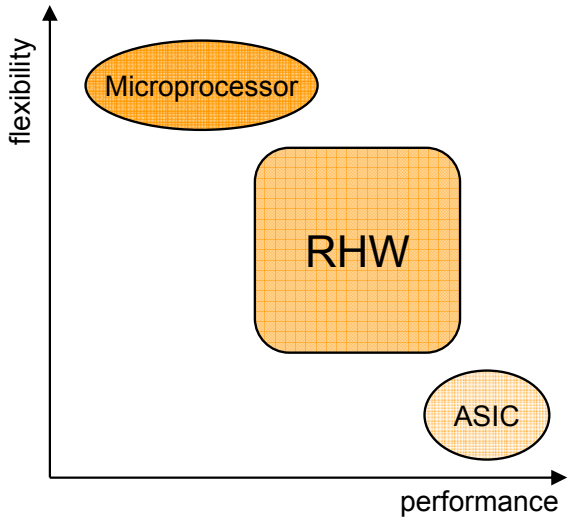

Figure 1.1. Classification of the different technologies in terms of flexibility and performance

Depending on the way this reconfiguration process is performed, several models can be identified. More specifically, there are three main criteria that allow classifying DRHW, namely granularity (coarse-grained and fine-grained reconfiguration), number of reconfigurations that each element stores (single-context and multi-context reconfiguration) and configuration style (total and partial reconfiguration). The next subsections describe these reconfiguration models in greater detail.

1.1.1 Fine-grained and coarse-grained DRHW

DRHW can be classified according to the granularity of the system, i.e. the size of the minimum reconfigurable element. Thus, we can find fine-grained and coarse-grained reconfigurable hardware.

Fine-grained reconfigurable hardware allows describing the hardware at bit-level; in other words, different operations can be assigned to each single bit of the system. In this configuration model, the basic logic elements are composed of look-up tables (LUTs) and flip-flops. For instance, for the latest Virtex™ devices, these LUTs have from 4 to 6 input bits and one or two outputs, which can implement any combinational function of the inputs. On the other hand, the flip-flops offer the possibility of implementing sequential systems.

In fine-grained reconfigurable systems, logic elements are also connected to each other at bit level; hence this type of granularity is very suitable for bit-level computations but might not be efficient for computations with word-width data. Thus, this configuration model allows obtaining the maximum flexibility, since an optimal configuration of the available resources can be created for each algorithm. However, this flexibility is not free: a price is paid in terms of area, length and number of existing interconnections, number of bits needed to load a reconfiguration, energy consumption and, in many cases, even performance.

In contrast with fine-grained reconfigurable hardware, we find coarse-grained reconfigurable hardware. In this configuration model, the bulk of the logic operations are performed at word-level (which usually ranges from 4 to 128 bits). In this case, the logic elements are usually Arithmetic Logic Units (ALUs), as well as a number of interconnection elements and storage resources. These ALUs can perform different operations, such as integer arithmetic, bitwise logic and bit-shifting operations, depending on the particular coarse-grained device. In addition, they can obtain their operands from different sources, depending on the information stored in a configuration register.

This type of granularity provides better performance and less energy consumption than fine-grained reconfigurable hardware, as long as the given task can conveniently adapt to the word size of the particular coarse-grained platform. In addition, the number of configuration bits needed to configure a task is much smaller using coarse-grained DRHW than using fine-grained DRHW (this usually depends on the flexibility of the particular coarse-grained platform).

However, this increase in performance comes at the price of losing flexibility, which is a great disadvantage in many dynamic contexts. This has prevented their widespread use in commercial devices. Hence the coarse-grain reconfiguration model exists mainly in the academic world. For instance, even though this model has been adopted in several commercial systems, such as Morphosys [SLL+00], RaPiD [ECF96] and MATRIX [MDH96], it has not had a great success among the users.

Among the existing reconfigurable devices, fine-grained reconfigurable architectures, more specifically Field Programmable Gate Arrays (FPGAs) clearly dominate the market. The main reason is that they are the result of many years of development and testing efforts, making them a mature technology, also supported by several development tools. In addition, in spite of being fine-grained reconfigurable architectures, many of them include coarse-grained elements, such as embedded multipliers or processors, which offer a good performance and simplify the design process. A great advantage of this mixed approach is that they allow using only the processors while the performance requirements are low and to use the reconfigurable resources only to meet the deadlines of the most demanding applications. Therefore, such platforms can achieve an average energy consumption comparable to a low-power embedded system, and provide the peak performance that many current applications demand. For instance, the latest Xilinx™ Virtex[1,2] and Altera™ Stratix[3] FPGAs implement this mixed reconfiguration model.

These are the reasons I have used FPGAs to carry out this work and to evaluate the results. More specifically, I have used a Xilinx™ Virtex-II Pro XC2VP30, which was the latest FPGA included in the Xilinx™ University Program when I started this research work, in September 2007. However, the developed work is fully scalable to bigger and more modern platforms, such as Virtex-5 or Virtex-6 FPGAs.

[1] *Virtex-5 FPGA User Guide*: http://www.xilinx.com/support/documentation/user_guides/ug190.pdf
[2] *Virtex-6 FPGA User Guide*: http://www.xilinx.com/support/documentation/user_guides/ug360.pdf
[3] *Stratix V Device Handbook*: http://www.altera.com/literature/hb/stratix-v/stratix5_handbook.pdf

1.1.2 Single-context and multi-context reconfigurable hardware

The majority of reconfigurable devices use a configuration memory (usually a SRAM) to store the configuration bits. These bits specify which hardware elements are used in a given reconfiguration and the way they are interconnected with each other. Thus, changing the content of this reconfiguration memory leads to a change in the functionality of the device. Although there is only one reconfiguration actually loaded in the DRHW, each logic element can be associated with one or several configuration bits. Depending on this feature, we can find single-context and multi-context reconfigurable hardware.

On the one hand, single-context reconfigurable architectures store just one configuration per logic element. Hence, every time a new reconfiguration has to be loaded in the DRHW, the proper information will have to be written in the configuration memory. This leads to significant time penalties whenever a new task is loaded in the system.

On the other hand, in multi-context reconfigurable architectures [FFM+99, SUA+00], the reconfiguration memory stores several configurations per logic element. In this case, the reconfiguration memory stores several contexts, which can be seen as planes that specify different functionalities of the device. There is only one active context, whereas the rest of them are stored in the reconfiguration memory. The user can switch from one context to another. For that purpose this kind of devices includes a control logic (a multiplexer, for instance), as Figure 1.2 shows. To simplify this control logic, the reconfigurable device must select *all the information that is stored in the same context*. Hence whenever the active context changes, the whole device is reconfigured. A context switching is performed in just one clock cycle, which is the main advantage of this configuration model with respect to the single-context approach. However, such devices require as many times the configuration-SRAM size needed in single-

context DRHW as the number of contexts, and caching tasks on the contexts consumes a large amount of static energy.

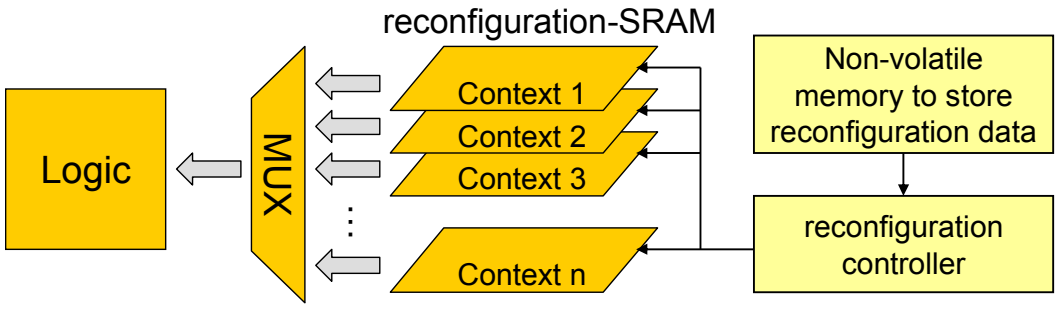

Figure 1.2. Multi-context reconfiguration model

This is the reason the majority of the commercial FPGAs include support for just one context. As they are fine-grained architectures, the number of existing reconfiguration bits for only one context is considerably large. Hence, adding support for several contexts would imply an enormous penalty in terms of configuration memory consumption, which is usually unaffordable for most manufacturers. As a result of this, multi-context FPGAs only exist in the academic world.

On the other hand, coarse-grained architectures use fewer configuration bits; hence in this case it is more affordable to include several configuration contexts. As I have based this research work on FPGAs, the target device that I have used only includes one reconfiguration context.

1.1.3 Total and partial reconfiguration

Depending on whether it is possible to alter the functionality of just a part of the device or not, we can characterize total and partial reconfiguration.

When the only way to load a task onto a reconfigurable platform is by completely reconfiguring the device, we are talking about total reconfiguration. I

have already mentioned this kind of reconfiguration when I introduced multi-context reconfigurable hardware. In these systems, it was not possible to load tasks belonging to different contexts at the same time. Hence, a context switching leads to a total reconfiguration process.

On the contrary, when it is possible to alter the functionality of just a subset of the available resources, while keeping the remaining ones unaltered, we are talking about partial reconfiguration. This is possible because the configuration-SRAM is actually constructed as a RAM. Therefore, the user can selectively change the content of a given portion of the configuration memory. Partial reconfiguration has two main advantages: on the one hand, it allows the execution of several independent tasks. On the other hand, thanks to partial reconfiguration, the configuration of a task takes a much shorter time than a total reconfiguration of the device.

Currently, the vast majority of the manufactures (such as Xilinx™, Actel™, Atmel™ or Altera™) include partial reconfiguration in their devices. However, the way this partial reconfiguration process is performed often differs from one device to other. For example, for Virtex devices, there are two basic ways to perform it: 1-dimensional (1-D) and 2-dimentional (2-D) models. For the 1-D model, the minimum set of resources to be reconfigured comprises a whole column of logic elements. Thus, the area occupied by a partially reconfigurable region is specified in terms of columns, and this area can span to its right or its left. Xilinx™ Virtex-II Pro FPGAs implement this model. On the contrary, the 2-D model eliminates this constraint, thereby allowing a partially reconfigurable region to be a rectangular region which height is not necessary a full column of reconfigurable resources. For their part, Xilinx™ Virtex-4 and Virtex-5 FPGAs implement this model.

The work that I have developed in this doctoral thesis assumes that the target device supports partial reconfiguration. Then again, it is compatible with both 1-D and 2-D reconfiguration models.

1.2. Challenges in the utilization of the reconfigurable hardware

Reconfigurable computing has caught the eye of many research groups and manufacturers in the past two decades. However, even nowadays we must consider that this novel technology is still in its infancy. In fact, since its first appearance in the 1980's to our days, the utilization of DRHW has forced designers to face a number of challenges, many of which still remain a major issue nowadays. This section presents an overview of the most widespread ones and puts in context the challenges that have been addressed in this doctoral thesis.

One of the main challenges of using reconfigurable devices has typically been the lack of supporting methodologies and tools at a high abstraction level in the design phase. For instance, given a system capable to execute both hardware and software tasks, it is not trivial to find a feasible hardware/software partitioning of the tasks to be executed so that the computations are wisely divided between hardware and software resources. This problem has been targeted by several research groups, and some relevant works have been proposed in [NB01, NB02a].

At a lower level of abstraction, and once a good hardware/software partitioning has been found, another issue is how to manage the tasks assigned to DRHW. In a DRHW-based multi-tasking system, several tasks are intended to run in parallel, thereby being assigned to different subsets of reconfigurable resources. However, in order to take advantage of the hardware multi-tasking, there must be some kind of execution framework to specify if a task can be placed anywhere in the available area or, on the contrary, if there are some fixed regions onto which they must be mapped. This usually has a direct influence on the way the communications are performed; hence multi-tasking systems must include a communication infrastructure to efficiently perform the interchange of

data among tasks. In this regard, there are many solutions proposed in the literature, and I will make an overview of the most relevant ones in the *Related Work* chapter.

Depending on the infrastructure of a given hardware multi-tasking system, different additional problems will raise, such as communication bottlenecks, fragmentation of the available space… In any case, common problems that every multi-tasking system has to address are the mapping, placement and scheduling of the tasks. Hence, such a system requires specific support to address these problems, which are usually intertwined. This means that, in order to reach a good scheduling, the mapping and placement constraints must be taken into account as well. Similarly, in order to reach a good mapping and placement, the scheduling must be also considered.

Regardless of the way tasks are assigned to reconfigurable resources, another well-known problem of using DRHW is the reconfiguration overhead related to the reconfiguration process. In order to load a task into a reconfigurable device, the proper configuration bits must be written in the configuration memory. For instance, for a Xilinx™ Virtex-II XC2V3000-4, these data comprises 15,868,192 bits and is fully configured in 24.3 ms[4]. This process takes time and energy, and can greatly degrade the performance of the system. It is important to underline that this is a major issue only when many reconfigurations are performed in a short period of time. However, if much time usually is elapsed between two consecutive reconfigurations, the reconfiguration overhead can be ignored, because it is relatively very small compared to the execution time of the task.

The work presented in this doctoral thesis addresses the latter challenge. Thus, I propose a task scheduling algorithm for a hardware multi-tasking system that explicitly takes into account the reconfiguration overhead throughout the scheduling process. It is a hybrid approach, since it consists of a design-time phase and a run-time one. In addition, I also propose an efficient hardware

[4] *Virtex-II Platform FPGAs: Complete Datasheet*: http://www.xilinx.com/support/documentation/data_sheets/ds031.pdf

implementation of the run-time phase of the scheduler using the available resources of the target Virtex-II Pro FPGA. Next section presents the target architecture, whereas Chapters 4 and 5 present more details about this scheduling flow and its hardware implementation, respectively.

1.3. Our work environment

The applications that have been used in this doctoral thesis include one or several control threads that are executed somewhere in the target system (for instance, in an embedded processor). These threads deal with dynamic events that trigger the execution of one or several tasks. Figure 1.3 shows two examples of control threads that deal with events that are generated at run time, which trigger the execution of different applications. We assume that these tasks are represented with Directed Acyclic Graphs (DAGs), which are a common representation used in embedded systems. For instance, a 3D game application may identify at run time that it needs to display a 3D object. This typically involves decompressing the object information, carrying out a computing intensive rendering process, applying some specific shading and texturing improvements, and finally performing a rasterization phase. If we assume that each phase is carried out by a different task, five tasks will be needed, and their data dependencies will be easily represented as a DAG.

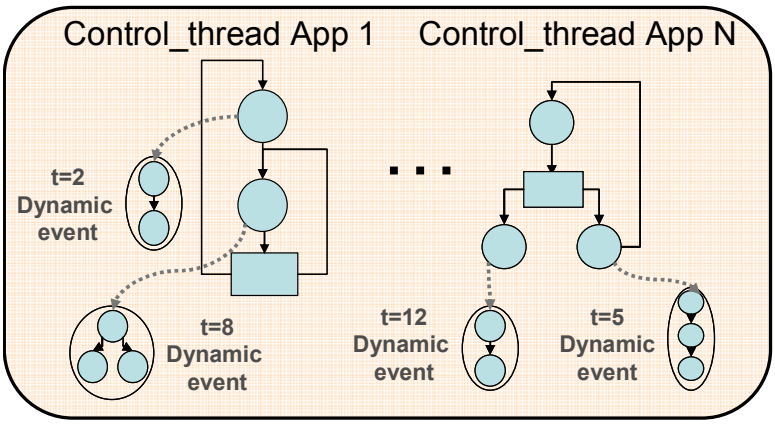

Figure 1.3. Execution scheme used

In these graphs, the nodes represent computational tasks and the vertexes represent internal task dependencies. A similar representation model has been successfully applied to represent complex MPEG-4 and MPEG-2 applications for embedded multiprocessor platforms [WMY01, YC04]. A hardware multi-tasking system seems an ideal choice to take advantage of the parallelism of these graphs, provided that the system includes the support needed to efficiently manage and schedule their execution.

Figure 1.4 presents the target architecture. The region labelled as "*Rec. HW*" includes the hardware implementation of the task-graph scheduler, as well as a set of reconfigurable units (RUs) of equal size. All of them are wrapped with a fixed interface that provides support for inter-task communications. Hence a hardware task can be physically placed in any RU and can communicate with all the remaining tasks in the system.

All these RUs are connected by means of a fixed "*Interconnection Network*". In the development of this doctoral thesis, it has been assumed that this infrastructure is contention-aware and provides enough bandwidth so that even the worst-case communication is performed correctly and efficiently. This can be achieved, for instance, by means of a bus with enough bandwidth or a contention-aware Network-on-Chip (NoC) [DMB02] that implements a *wormhole* routing algorithm or any similar technique that guarantees that the latency of a transmitted message between two whatever nodes is almost constant, regardless of the distance between them. Hence, under these assumptions we can safely assume that any task can be safely implemented on any of the RUs.

The reconfigurable part of our target system only includes one reconfiguration circuitry, as all the commercial FPGAs that exist in the market. This means that it can perform only one reconfiguration at a time, and hence it is not able to carry out several reconfigurations in parallel. In addition, it supports partial reconfiguration in such a way that this process does not interfere whatsoever in the execution of the remaining tasks already loaded in the system. This

decouples the reconfigurations and executions of the tasks, making them independent to each other.

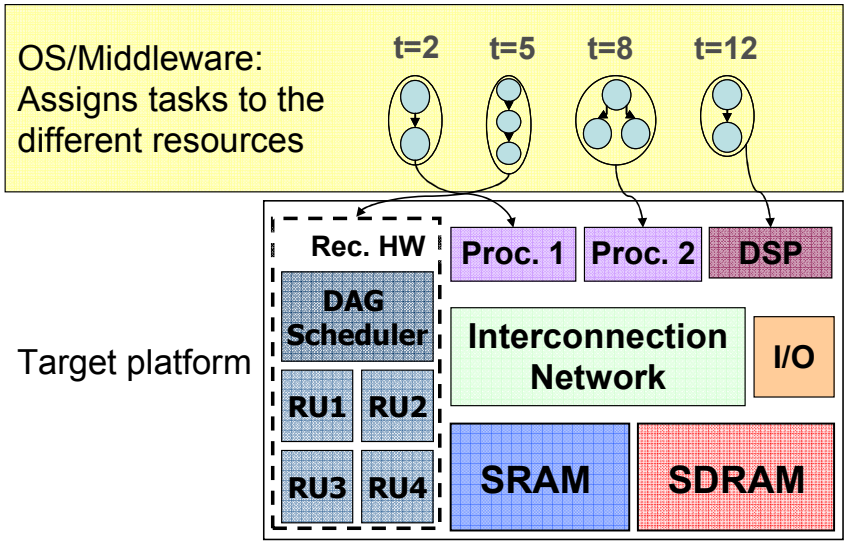

Figure 1.4. Target architecture

The "*DAG Scheduler*" explicitly takes into account these two points in order to achieve good scheduling results by performing the task reconfigurations and executions in parallel whenever it is possible. This technique is known in the literature as prefetch, and its benefits have been comprehensively studied in [Hauc98, LH02].

As Figure 1.4 shows, the target architecture also includes another processing elements (processors and DSPs), SRAM and SDRAM blocks that constitute its memory hierarchy, and Input/Output (*I/O*) controllers. Controlling all these elements, there is an operating system (OS) or middleware that provides the information about the tasks to be executed and, depending on their temporal constraints, decides whether a task must be assigned to an embedded processor, to a DSP or to a RU. And in the latter case, the task scheduler will attempt to execute them as efficiently as possible, minimizing the impact of their reconfiguration overhead.

The limits of my language mean
the limits of my world

Ludwig Wittgenstein

Chapter 2:
Background and contributions of this doctoral thesis

This chapter presents the motivation of this doctoral thesis as well as the contributions of this work. First of all, Section 2.1 underlines the need of run-time scheduling support in a hardware reconfigurable system with a high level of dynamism. Then, Section 2.2 studies two relevant state-of-the-art scheduling techniques and presents an experimental analysis that measures the run-time overheads that they generate when they are executed on an embedded processor. Thus, Subsection 2.2.1 describes the studied techniques and Subsection 2.2.2 shows the testing platform built in order to obtain these results, which are shown in Subsection 2.2.3.

Finally, Section 2.3 describes in detail the contributions of this doctoral thesis. The first contribution presented is a scheduling algorithm to target DAGs in the

architecture that was depicted in the previous chapter. This algorithm is a hybrid design-time/run-time approach. The other main contribution presented is an efficient hardware implementation of the run-time phase of the aforementioned scheduler, which goal is to minimize the penalties generated by its computations. Subsections 2.3.1 and 2.3.2 explain both contributions in detail.

2.1. Motivation

In the last few years, there has been a great growth in the performance requirements of many applications. Among these we can highlight standard multimedia applications, such as JPEG, MPEG and MHEG [HW90, Noll97, Mark92]. In fact, lately portable devices have included support for a great number of applications, such as video/audio coders and decoders, and applications that render 3-D graphics and that implement standard wireless connections. In a personal computer these applications are optimized including sound and video cards, graphics accelerators and wireless cards, which usually include ASICs to achieve the required performance. However, the lack of flexibility of these specific circuits makes them inappropriate in many contexts, especially in embedded systems, where usually there is not enough available space to include all these elements. In addition, the short time-to-market that these applications demand and the usually high manufacturing price of these specific accelerating elements are making them less and less attractive from the commercial point of view.

Hence, DRHW in general, and FPGAs in particular, have emerged in the last two decades as a promising technology that is able to cope with the dynamism and the high-performance requirements of many of these applications. Including hardware resources makes this technology powerful enough to carry out enormous amounts of computations efficiently, in the same way as an ASIC does. In addition, dynamic reconfigurability makes possible to target in the same device a very heterogeneous set of applications. This is the key feature that allows mapping in a single device a *virtually* unlimited amount of hardware tasks. (In practice, the *actual* number of applications that it is able to handle concurrently will be limited by the amount of available resources at a given moment of time).

However, nowadays manufacturers of reconfigurable platforms do not provide a work environment so that users can easily take advantage of the opportunities that DRHW offers. In fact, there is no specific commercial support for DRHW to allow the creation of tasks dynamically, or to manage the synchronization and communication among tasks and other elements in the system in a transparent way. Apparently, nowadays hardware multi-tasking is not interesting from the commercial point of view. However, it is evident that there exist in the industry numerous domains that can benefit from the use of this kind of systems.

For this reason, and in order to improve the usability of reconfigurable devices, there have been a great number of proposals of hardware multi-tasking systems by dividing somehow the total reconfigurable area into a number of tiles, not necessarily equal-sized, where hardware tasks can be executed. Thus, these systems are intended to take advantage of the parallelism of the targeted applications. However, they have been proposed so far only in the academic world.

In addition, and as I introduced in the previous chapter, one of the major drawbacks of using DRHW is that the reconfiguration time is usually very significant (in the order of hundreds of milliseconds). This means that, if these reconfigurations are performed too frequently, they could greatly degrade the performance of the system. Hence, a good scheduling strategy is essential to overcome this problem.

However, one of the main features of many current applications is their high degree of dynamism. In the execution context used in this doctoral thesis this means that, at a given moment of time, the complete sequence of task graphs that will be generated in the future is partially or totally unknown. Hence, at design time it is usually impossible to know which sequence of tasks will be executed in the reconfigurable resources.

This highly unpredictable behaviour makes difficult to make good scheduling decisions at design time. In this case, the only ways to deal with such degree of dynamism are: 1) to provide the system with enough resources to cope with the

worst-case scenario or 2) to guess which tasks will come in the future, according to some information gathered about previous task executions. Both solutions have great disadvantages: firstly, in embedded systems it is usually impossible to include such an amount of resources; and secondly, the predictions may be sometimes inaccurate, which leads to sub-optimal or even bad schedules. Hence, in this dynamic environment there is no alternative but scheduling, at least partially, at run time in order to make good scheduling decisions.

2.2. Experimental measurement of the scheduling overhead of previous relevant scheduling techniques

During the execution of a DAG in a multi-tasking system, normally an embedded processor will schedule its execution taking into account the internal dependencies of its tasks and the available resources. This involves executing complex scheduling algorithms, which deal with complex data structures at run time, and frequent hardware/software communications. As some of these heavy computations are performed at run time, the delays that their execution generates may also degrade the overall performance of the system (in addition to the overheads related to the reconfiguration process). Hence, these delays must also be minimized in order to prevent further performance degradations.

Before starting proposing and developing scheduling techniques for DRHW, we have carried out an experimental analysis of the overheads of some relevant techniques existing in the literature. We have selected the approaches proposed in [RMVC05] and [RMC05] since they were developed by our research group with the objective of obtaining very good run-time schedules, while generating a small run-time penalty due to the scheduling computations. However, such a detailed experimental measurement for embedded systems was still missing in these two works. The next subsection describes both techniques in greater detail.

2.2.1. The evaluated techniques

First of all, in [RMVC05] the authors propose a run-time reconfiguration scheduler to hide the reconfiguration latency in DRHW. The proposed scheduling flow is an extension of a previous multi-processor scheduler [YWM+01, YC03] in order to support DRHW. The goal of this extension is to provide support in order

to minimize the impact of the reconfiguration overhead. The new modules receive as input the schedule performed by the multi-processor scheduler presented in [YCM+01, YC03], identify the needed reconfigurations and attempt to reduce the reconfiguration overheads.

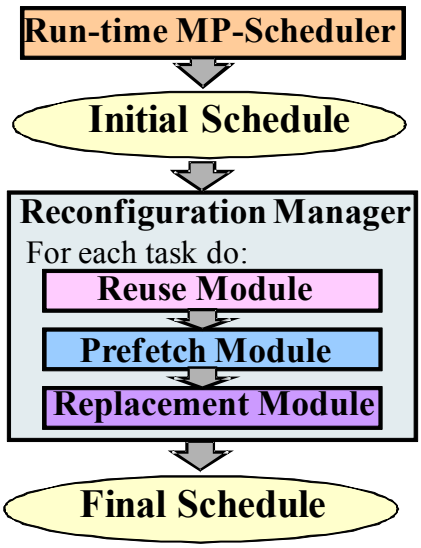

Figure 2.1. Scheduling flow presented in [RMVC05]

Figure 2.1 presents the different scheduling steps proposed in [RMVC05]. First of all, the run-time multi-processor scheduler analyzes the active DAGs and generates an initial schedule for them. This scheduler does not include the reconfigurations. Hence, in the following steps a "*Reconfiguration Manager*" updates the initial schedule in order to include them, and it also attempts to hide the reconfiguration latency as much as possible. The manager applies three techniques sequentially at run time: reuse, prefetch and replacement. First of all, the "*Reuse Module*" identifies which tasks can be reused from a previous execution. Then, if some tasks are not reusable, the "*Prefetch Module*" schedules their loads, attempting to minimize the execution time overhead. Finally, the "*Replacement Module*" decides where to load the incoming subtasks, trying to maximize the probabilities of reusing subtasks in the future. The main drawback

of this work is that the developed modules generate a significant run-time penalty due to the computations that they carry out. In fact, according to the presented experiments, this penalty is of the order of milliseconds, which is even comparable with the execution time of some of the proposed tasks. Hence, this is an important limitation since in some cases the performance of the system can be degraded simply due to the fact of carrying out the scheduling process at run time.

To overcome this problem, [RMC05] proposes a similar scheduling flow as in [RMVC05], but performing as many scheduling decisions as possible at design time, in order to save run-time computations.

The basic idea of this heuristic is that all the DAGs are analyzed at design time in order to generate an optimal schedule of the reconfigurations under certain assumptions for each one of these task graphs. Next, when one of them must be executed, the run-time scheduler checks if these initial assumptions are true. In that case, no further actions are needed. Otherwise, the run-time scheduler adds an initialization phase to the design-time schedule in order to guarantee that the initial assumptions are correct. Basically, the goal of this scheduling process is to minimize the run-time computations needed to hide the reconfiguration latency, since almost all the complex scheduling decisions are made at design time. In fact, the algorithm that generates the initialization phase only has a complexity of *O(N)*, where N is the number of reconfigurable units available in the system. According to their results, this hybrid design-time/run-time approach has proved to be almost as effective as the purely run-time one, since the reduction in the reconfiguration overhead that the hybrid technique achieves ranges from 0 to 1% worse than the purely run-time approach.

The techniques presented in [RMC05] were theoretically developed for embedded systems, but they were initially tested in high performance computers. Hence, the actual run-time overheads for embedded systems were unknown. However, this information is critical in order to know whether these techniques can be applied at run time for such systems or not. Hence we have carried out an

experimental measurement of the temporal overheads generated by these run-time scheduling techniques.

2.2.2. Testing platform

In order to evaluate the run-time temporal overhead generated by the selected techniques in embedded systems, we have run both schedulers in one of the embedded Power PC 405 microprocessors existing in a Virtex-II Pro XC2VP30 FPGA[5]. We have implemented the system using the Xilinx™ EDK environment[6], which provides basic components to easily develop an embedded system based on this processor. Figure 2.2 shows a scheme of the developed system.

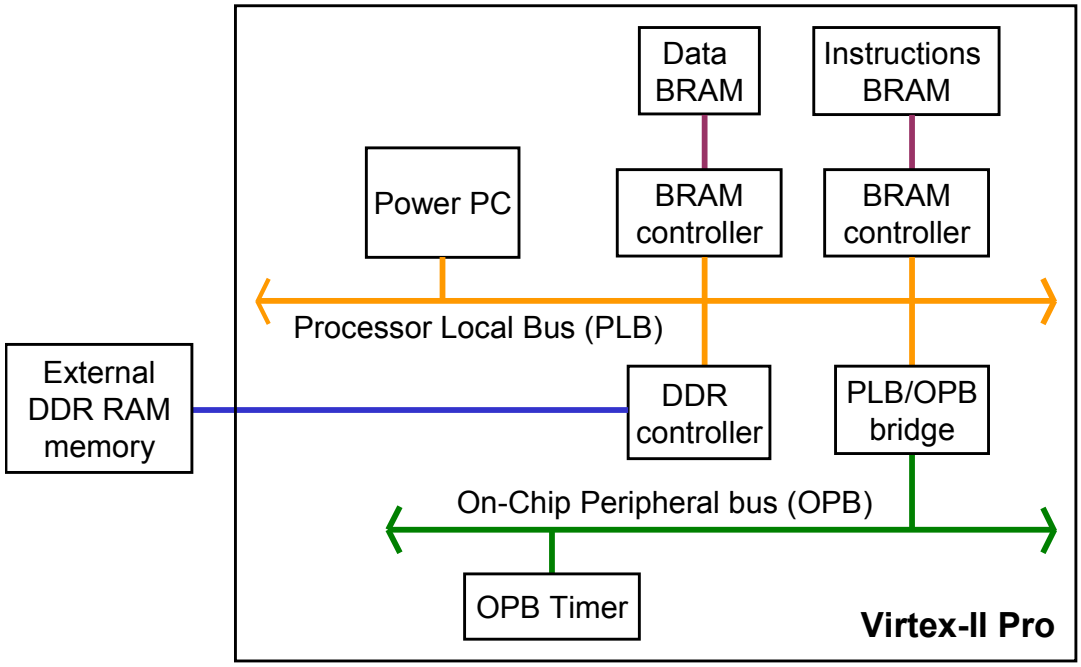

Figure 2.2. System developed in order to perform the temporal overhead measurements for the techniques presented in [RMVC05] and in [RMC05]

[5] http://www.xilinx.com/products/devkits/XUPV2P.htm
[6] http://www.xilinx.com/support/download/index.htm

The main element is of this system a Power PC that is directly attached to a "*Processor Local Bus*" (PLB). The two BRAM blocks are used to store the program and the data, respectively and they are attached to the PLB by means of a BRAM controller each. The processor has also at its disposal an off-chip memory (the "*External DDR RAM memory"*), which is attached to the PLB by means of a "*DDR Controller*". In addition, we have added a programmable counter (the "*OPB Timer*") in order to measure with clock-cycle accuracy how much time has elapsed during the execution of the tasks in this system. For that purpose, the processor communicates with this peripheral through a set of predefined orders that includes start, stop and resume commands. The OPB Timer needs to be attached to an "*On-Chip Peripheral Bus*" (OPB); hence the system includes a "*PLB/OPB Bridge*" so that the processor is able to reach it.

2.2.3. Experimental results

First of all, we have executed the run-time approach [RMVC05] and the hybrid one [RMC05], for different numbers of reconfigurations, and compared the execution times of the algorithms storing the code on on-chip and off-chip memories. Figure 2.3 presents this comparison. If the scheduler stores the code in an off-chip memory its execution consumes from 2 to 5 milliseconds, when the number of reconfigurations to schedule ranges from 2 to 8. However, using an on-chip memory largely improves these results, reducing this penalty by 6.5 times. Nevertheless, the execution time of the run-time scheduler may be still unaffordable in many situations. For instance, the system needs 0.9 milliseconds in order to schedule eight reconfigurations, which is still significant with respect to the execution time of the considered applications (for instance, 26 milliseconds for Pocket-GL). Hence, it is clear that the amount of computations performed at run time has still to be further reduced.

Figure 2.3 also shows the execution time for the hybrid approach and compares it with the purely run-time one, for both on-chip and off-chip

implementations. These results show that in both cases, the hybrid approach achieves an average speed-up factor of 22 in their execution time.

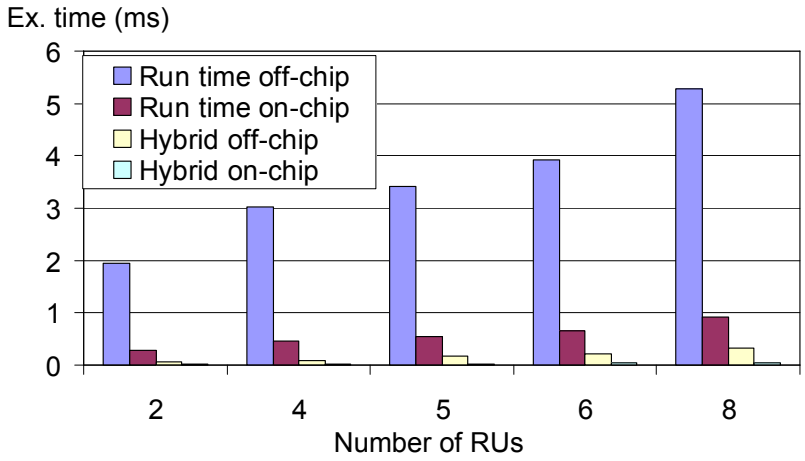

Figure 2.3. Execution times of the hybrid and the run-time schedulers

These results demonstrate that performing run-time computations involve important overheads in embedded systems. In fact, the simple management of the data dependencies among the nodes of a DAG can introduce important temporal overheads in the execution of the applications, which greatly degrade the whole performance of the system. A nice way to reduce these overheads is to move computations from run time to design time. If this can be done efficiently (as in the previous example), the overheads are greatly reduced, whereas it is possible to achieve comparable results in terms of performance with respect to a purely run-time approach. However, it is not always possible to move computations from run time to design time. Hence in this thesis I propose to move computations to design time whenever it is possible, and to speed up the remaining run-time phase by including specific hardware support for it. With this combination, it is possible to efficiently deal with really complex scheduling problems at run time.

2.3. Contributions of this doctoral thesis

This doctoral thesis addresses the problem of scheduling DAGs in the hardware multi-tasking system described in the previous chapter (Section 1.3). For that purpose, I have developed a scheduling algorithm to achieve an efficient use of the available resources, as well as a good performance for the targeted applications. In order to take into account the high dynamism of these applications, the algorithm performs an important part of its computations at run time. However, since we are looking for good-quality schedules without carrying out too many computations at run time, the proposed algorithm is a mixed design-time/run-time approach.

In addition, I have also developed an implementation of the run-time scheduler using some of the available resources in the target FPGA. Moreover, I have compared this implementation with a software one, and I have proved the convenience of using the hardware implementation rather than the equivalent software one, especially in terms of run-time penalties generated by the management/scheduling computations. Next subsections will describe these contributions in greater detail.

2.3.1. The proposed scheduling flow

This algorithm receives as input a set of applications represented as DAGs and schedules them in the set of reconfigurable units that our reconfigurable platform contains. The task graphs are executed in sequence. The information about each task graph comprises a set of nodes and the precedence constraints among them, which represent control and data dependencies, as it has been explained in Subsection 2.1. In addition, the nodes of each task graph also include an estimation of their execution time.

Our scheduler has been built to deal with just one task graph at a time. Hence, the execution of several task graphs using our scheduler is carried out sequentially. In addition, we assume that it works under dynamic conditions in which the only information that the scheduler has at its disposal is the task graph that is currently under execution and the dynamic status of the system. Therefore, in a given moment of time, the sequence of task graphs that are going to be executed in the future is completely unknown.

As described in Figure 2.4, the scheduler is composed of two basic modules: a **design-time scheduler** and a **run-time one**. Basically, the **design-time scheduler** receives the basic information about the incoming task graphs and analyzes each one of them separately in order to extract some useful information that will be used at run time. This information characterizes each task with three parameters, namely **weight, criticality,** and **mobility**. Each one of them represents the following:

- The **weight** parameter is used at run time to decide the reconfiguration order. The idea is to reconfigure first those tasks that have a greater impact in the critical path of the graph.
- The **criticality** identifies the delays that the reconfiguration of each task may generate, and it is used to assign greater priority to the tasks that generate greater delays.
- Finally, the **mobility** is used to escape from local-optimum scheduling decisions delaying at run time some reconfigurations, taking into account the dynamic status of the system.

Figure 2.4 shows the updated task graph after executing the design-time scheduler. After this stage, the original graph is extended with new nodes, which in this case do not represent computing tasks but run-time reconfigurations, and

new edges, which represent the dependencies introduced due to the selected sequence of reconfigurations. Each task graph is also updated adding the information about the criticality and the mobility of the tasks (C_t and M_t for a given task t).

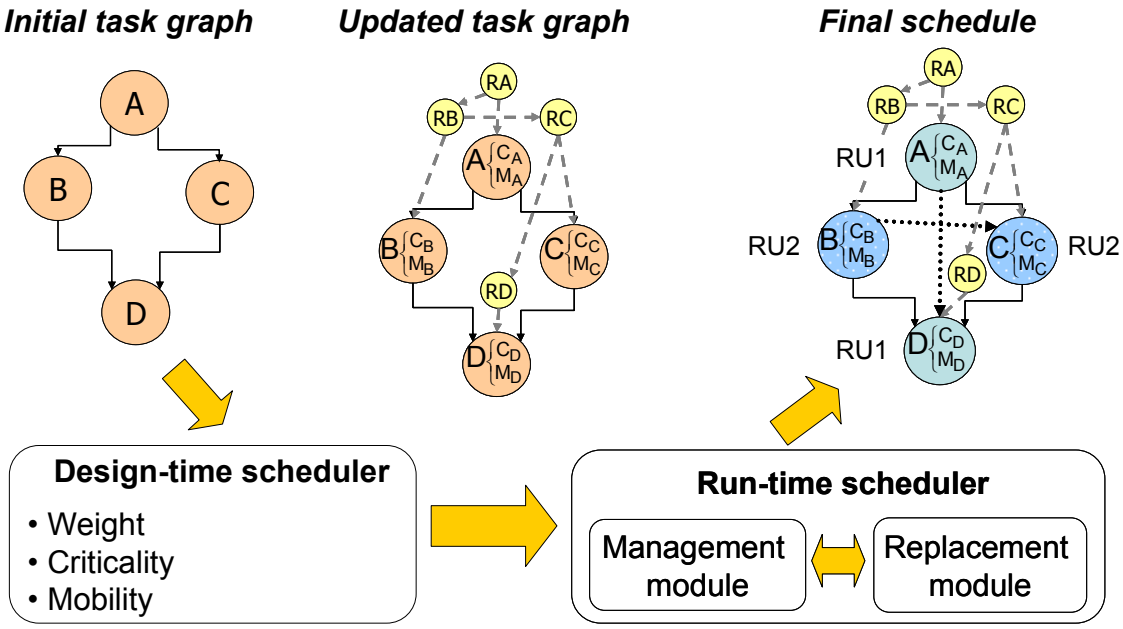

Figure 2.4. Scheme of the proposed scheduling flow

Once this information has been obtained, the run-time scheduler is executed. Basically, this part of the algorithm steers the execution of the task graph taking into account the information that was gathered at design time and, in addition, making some scheduling decisions in order to hide the reconfiguration overhead of the tasks. For this purpose, two basic techniques are applied.

- **Task prefetch**: The Virtex-II Pro FPGA used in this work is able to carry out a reconfiguration without interfering with the execution of the remaining tasks already loaded in the system. This means that it is possible to reconfigure a task in a RU while others are still running in the system. Hence, this feature allows loading a task in advance (i.e.

before those tasks are ready to be executed), overlapping its reconfiguration with the execution of others. This technique is known in the literature as task prefetch [Hauc98, LH02].

- **Task reuse**: In order to run a task in the reconfigurable platform, the reconfiguration of that task must be carried out in advance. However, it is not necessary to carry it out if the involved task already exists in the system because in that case it can be reused. If applied wisely, and using a good replacement policy, the task reuse can offer clear benefits in the performance of the system when dealing with recurring tasks, as will be explained in the following chapters. In our scheduling flow, this technique collaborates with the prefetch in order to multiply their respective advantages.

The **run-time scheduler** completes the scheduling process and returns the final schedule of the incoming task graphs. Thus, for instance, according to Figure 2.4 the run-time scheduler has assigned tasks A and D to RU1, and task B and C to RU2. In addition, the schedule specifies that A will be executed before D, and B will be executed before C. Due to this assignment two new edges are added to the original graph (from A to D and from B to C) in order to guarantee that the task-graph execution will follow the given schedule and to prevent structural conflicts. It is important to remember that our scheduler processes the incoming task graphs in sequence, not in parallel.

As shown in Figure 2.4, this phase is in turn divided into: **management** and **replacement modules**. Both of them collaborate closely with one another in order to obtain the final schedule.

The **management module** steers the execution of the task graph taking into account its precedence constraints, the available resources and the replacement decisions given by the replacement module. In order to take all these run-time decisions as quickly as possible the scheduler only considers some discretized time instants following an event-triggered approach. When certain events happen

the scheduler will look for reconfigurations of tasks that are ready to be scheduled, trying to load or execute them *As Soon As Possible* (ASAP). Hence, the prefetch is applied in this moment. In addition, under certain conditions some reconfigurations will be delayed, depending on the mobility of the tasks and the current status of the system. Hence this is actually a modified ASAP approach.

For its part, the **replacement module** collaborates with the execution manager to decide where to load a task. The reuse technique is applied in this moment: first of all, the replacement module checks whether the task can be reused in any RU or not. If so, the task will be reused and no reconfiguration overhead is generated. Otherwise, it applies a replacement heuristic in order to select the target RU, replacing the task that was loaded there as a result of a previous execution. This decision is taken depending on the criticality of the tasks already loaded in the candidate RUs, and whether the candidate is going to be executed again in the future or not.

This PhD dissertation also includes a comprehensive comparison between this approach and some well-known scheduling algorithms proposed in the literature. In fact, the proposed scheduling algorithm has proved to outperform all of them. Chapters 4 and 5 of the present PhD dissertation describe in greater detail how this scheduling flow works and Chapter 6 presents the comparative results.

2.3.2. An efficient hardware implementation of the run-time scheduler

The second important contribution of this doctoral thesis is the implementation and testing of the run-time scheduler using a small amount of the reconfigurable hardware resources existing in the target FPGA. The goal is to perform the heavy run-time computations as fast as possible, thereby generating just a small run-time penalty (in this particular case, just a few clock cycles). To test this hardware

module a simplified simulation environment has been developed and implemented on the FPGA. This environment uses a set of programmable timers to simulate the behaviour of the RUs. Hence this platform does not perform partial reconfiguration on the reconfigurable device, but it simulates the reconfiguration and execution times of the tasks. However, this is the only feature that is simulated in the whole system. This system also includes support to measure with clock-cycle precision the overheads generated by the scheduler. Hence it has been used to evaluate the performance of the scheduler.

In addition, this doctoral thesis makes a comparison of the hardware implementation of the run-time scheduler with an equivalent software one, which consists in a program written in the C programming language and compiled for one of the embedded Power PC 405 microprocessors existing in the Virtex-II Pro. According to our results, the software version has proved to be up to three orders of magnitude slower than the hardware one. However, it has the advantage of not using any reconfigurable resources. Hence, these two versions offer different trade-offs between the run-time scheduling overhead and the cost needed to implement the hardware circuit.

More details about the proposed architectures, as well as a comprehensive study of the features of both of them, can be found in Chapter 5 of this PhD dissertation.

There are two great rules of life: never tell everything at once

Ken Venturi

Chapter 3: Related work

Reconfigurable computing has caught the eye of many research groups and manufacturers in the past two decades. Hence, in order to properly evaluate the work developed in this doctoral thesis it is convenient to comment the current status of other related research works. In addition, it is essential to highlight the main differences between my work and other approaches in the reduction of reconfiguration overheads. Instead of providing a complete survey of reconfigurable computing, this chapter focuses only on the research that is most related to our work. Exhaustive surveys can be found in [TCW+05, AA09].

Thus, Section 3.1 starts commenting different contributions proposed in the literature towards the implementation of a hardware multi-tasking system using DRHW. Next, Section 3.2 presents some contributions that extend the functionality of an operating system in order to make it compatible with DRHW.

Section 3.3 gives an overview of the state-of-the-art techniques proposed by different research groups to tackle the problem of reconfiguration overheads and relates them to the work presented in this PhD dissertation. Finally, Section 3.4 finishes this chapter with some final conclusions.

3.1. Hardware multi-tasking support for dynamically reconfigurable systems

One of the greatest advantages of using dynamically reconfigurable devices as development platform is the possibility of taking advantage of the parallelism of a hardware design. However, up to now, none of the manufacturers has included support to easily take advantage of the profits of general multi-tasking. This is the reason many efforts have been done to enable general hardware multi-tasking in order to hide many low-level details to the user, such as, for instance, communications management or partial reconfiguration.

This section makes an overview of some the most relevant works proposed so far in the literature about this topic. First of all, Section 3.1.1 summarizes the work of T. Marescaux et al., which present a series of contributions towards the development of a customized an efficient Network-on-Chip (NoC) [DMB02] to enable fine-grained multi-tasking on DRHW, which is controlled by an operating system. Next, Section 3.1.2 comments the work of H. Walder and M. Platzner, a partially reconfigurable architecture built to be deployed in a Virtex-II FPGA. Finally, Section 3.1.3 describes the BEE2 architecture, an FPGA-based reconfigurable computer developed at the Berkeley Wireless Research Centre.

3.1.1. Work of T. Marescaux et al.

One of the first contributions in the field of DRHW proposing an actual implementation of a hardware multi-tasking system was [MBV+02], presented in 2002. In this work, Marescaux et al. propose to build a fine-grained multi-tasking system by dividing the entire area of the FPGA into a set of equal-sized tiles wrapped with a fixed interconnection interface. This makes possible to decouple the communications and the computations, hence a reconfiguration of a task

does not interfere in the rest of the reconfigurable units, neither in the messages that the tasks are sending to one another. The main contribution of this work is an interconnection network and a routing protocol to perform the communications through it.

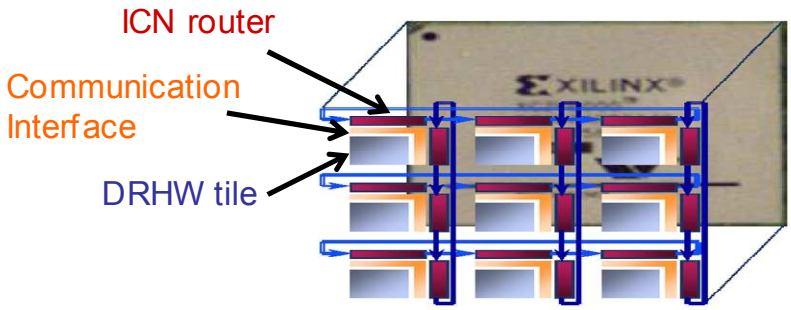

Figure 3.1. Scheme of the communication architecture proposed by T. Marescaux et al.

The proposed network is composed of a number of routers connected among them in a 2-D torus topology, as shown in Figure 3.1. With respect to a 2-D mesh, this topology has the advantage of its simplicity since the communications can be performed just by routing the packages in two directions (East and South), whereas a 2-D mesh requires communications to be routed in the four possible directions (North, South, East and West). Other big advantages of this approach are a better performance, since communications are distributed; less power consumption, since idle links can be dynamically powered-off; and scalability. In this work the authors also propose an implementation of a router belonging just to a 1-D torus, due to the limitations of the FPGAs that they used (a Virtex XCV800 and a Virtex XC2V6000). However, at that time this was already a significant breakthrough towards a better usability of DRHW.

In [MMB+03], presented in 2003, the authors continue the same line initiated in [MBV+02] by adding new features to overcome the limitations of the previous communication interface. The main contribution of this work is to enable task migration by creating two communication infrastructures that decouple different types of transmitted messages. The first one, also called **"Application Data**

Network" performs standard communications among tasks. And the second one, also called **"Control Network"** is used by the operating system to control the behaviour of the complete system. Thus, the latter allows data monitoring, task synchronization and debugging, and control of the hardware blocks.

On the one hand, the **"Application Data Network"** is built to guarantee high bandwidth, since much information may be required to be transmitted at the same time. For that purpose, it implements a communication load control system, which limits the amount of messages that a task is allowed to send on the network per time unit. This is done to prevent errors in the transmission of the messages. In addition, this network also includes hardware support for the sake of the security of the system. For this purpose, it keeps track of the messages sent by any task in the system and, for each one of them, it checks whether the message length is smaller than the maximum transfer unit, or if the messages are delivered in order, among others. Otherwise, the communication is illegal. On the other hand, the messages transmitted through the **"Control Network"** are short, not necessarily too many, but they need to be delivered fast. Hence, rather than high bandwidth, this network needs to provide low latency. In this case, it is implemented as a shared bus. However, the communications on this bus are message-based and therefore can be easily replaced by any type of NoC, if necessary.

Finally, in [NMA+05] the authors improve the infrastructure of [MMB+03] by including a task assignment heuristic to map tasks in the target system, which includes both software processors and DRHW tiles. It consists in a basic algorithm that contains ideas from multiple resource management algorithms [KSS+03, HM04] and that is complemented with reconfigurable add-ons to take the DRHW properties into account. The first one is the consideration of internal fragmentation of reconfigurable area. The second one involves hierarchical configuration, in such a way that the system may choose to instantiate a soft-core in a hardware tile and deploy a software task there rather than in a processor. The reason of this may be the inexistence of an available software processor in the system at that moment, or that the available processors do not provide

enough communication performance with the remaining deployed tasks. In addition, this work also proposes a task migration mechanism that allows moving an application from a given source tile to a destination one at run time. This migration mechanism includes support to ensure message consistency, for instance if the involved task was sending/receiving data in the moment of the migration request.

This research group has published more recent works about significantly increasing the quality of service (QoS) on networks-on-chip in [Mare07, MBC07]. However, these works no longer care about DRHW multi-tasking itself, but about the performance of the communication layer, which is out of the scope of this doctoral thesis.

3.1.2. Work of H. Walder and M. Platzner

Another interesting work in this field was developed by H. Walder and M. Platzner in [WP04]. In this work, the authors present the design and implementation of a runtime environment for generic tasks in DRHW. For that purpose, they split the FPGA into a static and a dynamic region. As shown in Figure 3.2, the static part comprises the operating system (OS) modules and it is organized into two OS frames located at the left and right edges of the FPGA. For its part, the dynamic region comprises logic resources available for user hardware tasks, which are dynamically loaded. This region is divided into several slots that can accommodate the hardware tasks.

At the time this work was developed, the 2-D reconfiguration model had not been implemented yet in FPGAs, hence they used a device that supported just a 1-D partial reconfiguration model: a Xilinx™ Virtex-II XC2V3000-4[7]. Thus, these slots span all the height of the reconfigurable device. Although the number of

[7] *Virtex-II Platform FPGAs: Complete Datasheet*:
http://www.xilinx.com/support/documentation/data_sheets/ds031.pdf

slots is fixed and equal-sized, a big task can use several consecutive slots if necessary.

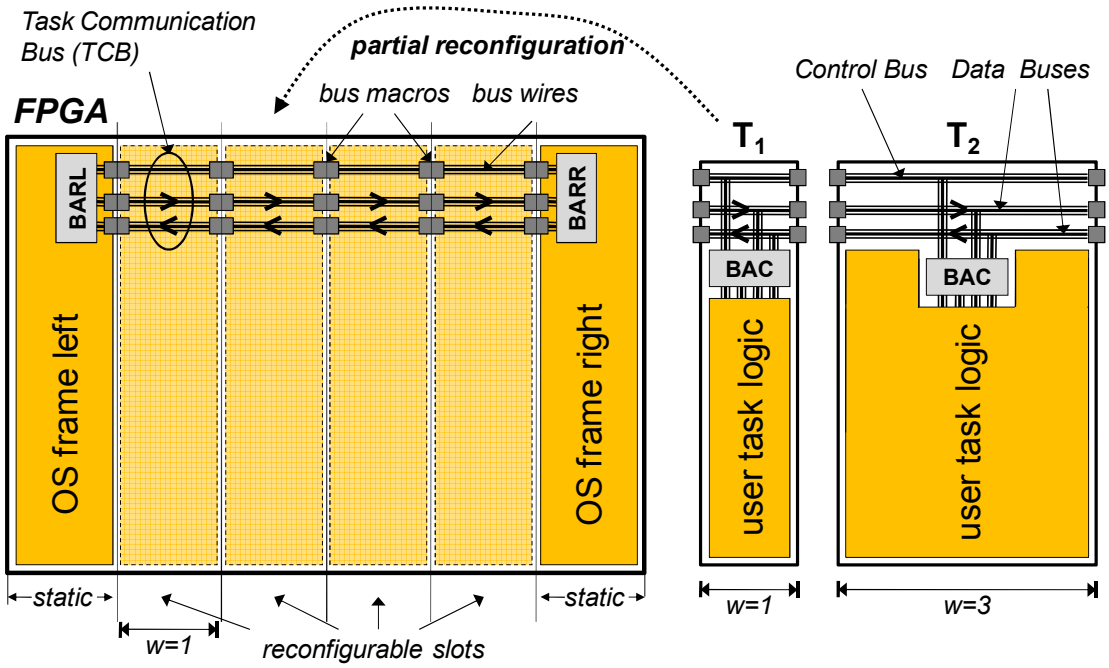

Figure 3.2. Block Diagram of the system proposed by H. Walder and M. Platzner

All the slots are interconnected among them by means of a "*Task Communication Bus*" (TCB), which in turn is divided into a "*Control Bus*" and two additional "*Data Buses*". The latter ones perform communications from left to right and from right to left, respectively. In addition, the TCB is ruled by two bus arbiters, "*Bus ARbiter Left*" (BARL) and "*Bus ARbiter Right*" (BARR), as Figure 3.2 shows. They are located in the right and left OS frames, respectively.

In order to connect the TCB with the reconfigurable slots, each one of them must include bus macros and some additional "*Bus ACcess logic*" (BAC). On the one hand, the bus macros allow establishing the routing between the static and the reconfigurable parts, making the communication ports of the reconfigurable slots compatible with the static part of the system. Hence all the connections that are driven from/towards a reconfigurable module must pass through a bus

macro. On the other hand, the additional "*Bus Access Logic*" handles the bus reservation and implements the data transaction protocol. Thus, the actual bus protocol is hidden from the user's point of view.

3.1.3. The BEE2 project

The Berkeley Emulation Engine 2 (BEE2) [CWB05] is the last relevant research work in this field that I want to comment. This project pursues a reusable, modular and scalable multi-tasking system. The proposed framework is a FPGA-based reconfigurable computer built at the Berkeley Wireless Research Center (BWRC) around year 2003. This project was born because they needed a platform capable of targeting complex applications with real-time constraints and at minimal power consumption. Among these applications there were sophisticated encoding-decoding, software-defined-radio design, cognitive spectral reuse, and multiple antenna MIMO (multiple-input, multiple-output) algorithms.

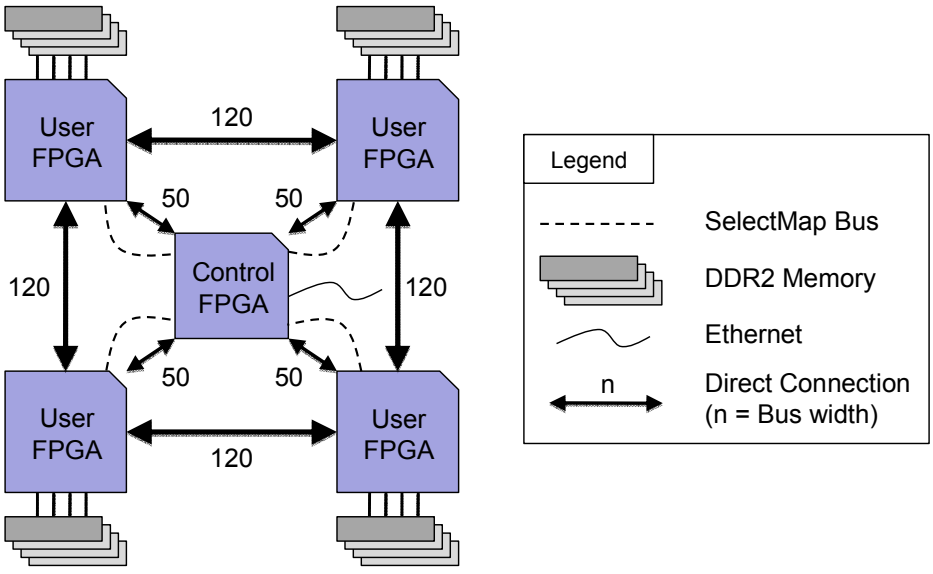

Figure 3.3. Block Diagram of the BEE2 architecture

Figure 3.3 shows a high level block diagram of the BEE2 architecture. Each BEE2 compute module contains five Xilinx™ Virtex-II Pro XC2VP70 FPGAs. On the one hand, the central FPGA, usually referred to as the "C*ontrol FPGA*", is connected to each of the remaining four "*User FPGAs*" through an 8-bit SelectMap bus and a 50-bit direct connection. The "*Control FPGA*" carries out the system functions, such as configuring the four "*User FPGAs*" and the network communications. On the other hand, the four "*User FPGAs*" are connected in a ring topology through 120-bit direct connections.

In addition to the people from Berkeley Wireless Research Center, other research groups have used this platform to develop their research, such as in [TCB06, MRS+07]. In addition, this has been the development platform used by H. Kwok-Hay So for his research [SB08a, SB08b, STB06]. This work will be described in detail in Subsection 3.2.1 of this chapter.

3.2. Operating System (OS) support of DRHW

Other interesting contributions for hardware multi-tasking systems are the research efforts to develop OS support to simplify the use of the DRHW. This section comments several recent relevant contributions in this field. All of them share the same idea of hiding the hardware complexity behind a set of well-defined Application Program Interfaces (APIs). This constitutes a design contract: as long as applications adhere to the API, the OS guarantees that hardware-software interaction will have well-defined semantics.

[WK02] is an interesting discussion about this topic. This article points out the basic functionalities that an OS with specific support for DRHW should include:

- **Application loading**: The OS must manage the load and initialization of the tasks. This process involves placing the circuit and its embedded RAM on the FPGA; more specifically, in the exact position that the bitmap indicates. In addition, the load manager must map the necessary connections to guarantee the correct input/output of the tasks. The OS must also take into account that a hardware task starts its execution immediately after it has been loaded, whereas a software task does not behave in that way.
- **Partitioning and memory management**: A traditional OS usually manages a *virtual* memory space which is greater than the *physical* RAM that it actually uses. To perform this management, a software application is typically split into “pages”, which are stored in the hard disk and are loaded in the RAM if necessary. Hence, similarly to this management, an OS that includes support for DRHW should be able to somehow split a hardware task into several pieces. However, this

process is not trivial since a hardware circuit cannot be arbitrarily partitioned.

- **Scheduling**: When a task comes to the system, the OS has to decide when and where to load it. A good scheduling algorithm is a key feature so that the system fulfills the temporal constraints of the tasks and respects their priorities. In addition, the scheduler must efficiently manage reconfigurable resources, generating as low run-time penalty as possible.
- **Security**: The OS must prevent faulty or malicious tasks from getting loaded in the system. For instance, in a typical OS, a software process does not have access to the memory space in which another process is allocated. Similarly, a hardware task must not occupy the area reserved for other running tasks.

The following two subsections describe in detail two relevant works about this topic: BORPH, an operating system developed by H. Kwok-Hay So for his PhD (Subsection 3.2.1); and the work developed by K. Kosciuszkiewicz et al. (Subsection 3.2.2).

3.2.1. Work of H. Kwok-Hay So

The first relevant contribution in this field that I want to mention is the work developed by Hayden Kwok-Hay So in University of California, Berkeley [SB08a, SB08b, STB06]. The authors of this work propose an operating system for FPGA-based reconfigurable computers called BORPH (the Berkeley Operating System for ReProgrammable Hardware). This system extends UNIX semantics, well-known by most researchers, to reconfigurable computers. Thus, they provide the user with the capability of executing hardware tasks (that they called "*gatewares*") as if they were software processes. Hence, every time a *gateware*

is executed, BORPH creates a hardware process. Such processes contain not only the typical information of a software one, but also some spatial information about the reconfigurable resources that are being used. In many aspects, the hardware processes are managed similarly to the software ones: for instance, they may also be sensitive to UNIX signals (SIGTERM, SIGKILL…) and the input/output is managed in the same way as in a software process. However, there are some significant differences in the way communications are performed from/towards a *gateware*. On the one hand, the relationship between processor and user *gateware* designs is no longer master-slave: in BORPH an active communication may be initiated also by the *gateware* design, hence a hardware process can behave either as a master or a slave. On the other hand, there exist communications involving different types of processes (hardware – software, software – hardware, in addition to hardware – hardware). Including this feature in BORPH has forced the designers to implement a customized hybrid system call interface to send and receive messages.

BORPH is intended to be deployed on a BEE2 architecture, which, as mentioned above, is composed of four "*User FPGAs*" and a "*Control FPGA*". The "*User FPGAs*" are used as reconfigurable units to execute user *gatewares*, whereas the "*Control FPGA*" is responsible for system calls, configuring the remaining *"User FPGAs*" and performing network communications. Its kernel is divided into two parts: *mK* (main Kernel) and *uK* (user Kernel).

- ***mK*** is a modified version of a Linux 2.4.30 kernel and runs on one of the embedded Power PC 405 microprocessors existing in the "*Control FPGA*". Basically, it manages the software processes and implements the new features introduced to support the execution of files with extension *.bof*. *mK* is also responsible for creating and managing the *gatewares* at high level.
- ***uK*** is the part of the BORPH's kernel that manages the *gatewares* at low level. It is distributed over all the "*User FPGAs*", hence every time a

gateware is configured in a reconfigurable region, the corresponding part of this kernel must be reconfigured in the target FPGA as well.

In spite of all these interesting features, the latest version of BORPH still has some limitations. The most important ones are its low portability and that the management of the hardware tasks generates very significant delays. For instance, creating a hardware process can take up to 900 milliseconds. Hence there still is a lot of work to be done in order to develop an OS that can manage reconfigurable resources not only transparently, but also with a good performance.

3.2.2. Work of K. Kosciuzkiewicz et al.

Another interesting contribution in the field of OS research for DRHW has been proposed in [KMK07]. In the same way as the aforementioned BORPH operating system, this work also presents a modified LINUX operating system to manage the execution of hardware tasks.

The main novelty of this system is that each application is divided into a set of cores, and for each core the system manages a software and a hardware version of it. Thus, the system may change dynamically from the software version of a given core to the hardware one, depending on the dynamic status of the system. For that purpose, both versions need to share their external interface to the highest possible extent.

The described prototype runs on a Xilinx™ MicroBlaze processor[8] and the reconfigurable resources are modelled by eight dynamically reconfigurable PicoBlaze processors[9]. This means that this system do not manage actual hardware tasks, but it simulates their behaviour by means of these eight

[8] http://www.xilinx.com/tools/microblaze.htm
[9] http://www.xilinx.com/ipcenter/processor_central/picoblaze/picoblaze_user_resources.htm

processors. This prototype has been satisfactorily tested with the AES [DR02] and IDEA [LMM91] cryptographic algorithms on a Virtex-II Pro FPGA.

3.3. Elimination of the reconfiguration overhead: proposed techniques

The majority of the extensions for operating systems commented in the previous section have the objective of make possible the partial reconfiguration at run time. However, as introduced in Chapter 1, the high reconfiguration latency is still a major problem when using DRHW. The main reason for this is that the configurations are frequently stored in an off-chip memory and an external controller supervises the configuration process. In fact, this process can greatly degrade the performance of the system when many reconfigurations are carried out in a short period of time.

This is the reason many research groups have tried to solve this problem. An interesting survey of these techniques can be found in [PRMC07]. According to it, these efforts can be divided into three groups. The first group of techniques attempts to reduce the amount of required configuration data in each reconfiguration process. The second group proposes to reduce the number of required configurations. This is suitable for those applications that require running tasks repetitively. The last group takes the configuration process into account during the task scheduling in order to reduce its effect. All these techniques mainly focus on reducing the configuration latency, although for the second group, reducing the number of required configurations can also result in less dynamic configuration energy. The next subsections present an overview of these techniques and relate them with several relevant works.

3.3.1. Reducing the size of the configuration bitstreams

In the vast majority of the current FPGAs, the configuration information are presented as bitstreams, which consist of the configuration data to be stored in

the internal configuration memory as well as a set of instructions for the configuration controller. On the one hand, the configuration data configures the FPGA architecture; that is, the logic elements, the interconnection network, the input/output pins… On the other hand, all these instructions are sent to the reconfiguration controller so that it is initialized, the clocks are synchronized and the controller determines the memory addresses at which the data will be written. The format of a configuration bitstream depends on the features of the controller, as well as on the particular FPGA. As a result, bitstream formats vary among different vendors, even among different FPGA families of the same manufacturer.

The time needed to carry out a reconfiguration directly depends on the size of the bitstream that is to be written in the reconfiguration memory. Hence, a straightforward way to reduce the reconfiguration latency is by reducing its size. This can be achieved by applying techniques to **compress the bitstreams** or by **using architectures that require smaller bitstreams**, such as coarse-grained reconfigurable ones.

Many techniques have been proposed to **compress the amount of configuration data to be transferred**. In fact, many redundant information and regularities exist in the bitstreams and these techniques usually take advantage of this fact in order to remove the useless information. [LH01] makes a comprehensive study of several lossless data compression algorithms: the Huffman coding [Huff52], the arithmetic coding [Riss76] and the Lempel-Ziv-Welch (LZW) coding [ZL77]; as well as the viability of applying them to compress bitstreams for any SRAM-based FPGA. Probably the most popular one is the LZW. This technique creates a dictionary on-the-fly, which is conceptually composed of a set of trees representing the sequences of symbols that have appeared so far in the bitstream. The dictionary can be implemented by means of a table, which size grows as the bitstream is analyzed. Each entry of the table contains a symbol and the address where the previous symbol in that sequence has been stored in the table. Thus, a single entry of the table actually represents a sequence of symbols, since one can successively follow the addresses stored in the entries of the table and build a unique sequence. The brilliance of this

algorithm consists in being able to create the output by analyzing the input data only once, unlike other compression techniques. However, the main problem of the LZW algorithm is that large sequences of symbols with different small roots and similar big suffixes make the compressing algorithm inefficient, since the dictionary grows too much. Hence, some works have been proposed to extend this algorithm and to use it for bitstream compression as well. An interesting approach has been proposed in [DP05], which pursues better compression rates by codifying a commonly repeated suffix as if it was a single character. This algorithm achieves a memory saving that can be up to 11-41% of the original bitstreams.

However, these generic techniques do not consider the individual features within the reconfiguration bitstream of the particular device. Hence, they do not get even close to the theoretically optimal compression rate. This is the reason compression algorithms have been proposed for specific FPGAs, such as Virtex™ devices. For instance, in [GC08], the authors propose to adapt the extension of the LZW algorithm, which was initially published in [DP05], for Virtex FPGAs. Thus, they increase the compression rate of the bitstreams by 12% with respect to this previous work. The group of S. Hauck and Z. Li were also among the first ones to publish works about this topic. For instance, in [LH01, HLS98] they propose two compression algorithms for Virtex devices in general and for the Virtex XC6200 FPGA, respectively. The common idea is to compress the bitstreams, store them in the SRAM memory, and decompress them later on by means of a hardware module (such as a decompressor) that must be implemented using some of the available reconfigurable devices of the FPGA. Hence, these two techniques come at the cost of the hardware resources consumption that the decompressor needs, as well as the compression-decompression latencies. However, more recently (2009), the compression algorithm proposed in [SHWK09] describes a technique that is able to reduce the size of the bitstreams in modern reconfigurable devices (such as the Virtex-4), without the need of using any decompression circuitry. For that purpose, the algorithm eliminates the instructions that redundantly write zero frames to the

FPGA configuration logic from the bitstream. By doing this, this approach reduces the bitstreams up to just 3% of their original size and performs up to 4 times better with respect to the bitstream compression feature that is included in the Xilinx™ ISE tool (–g option).

Another approach to reduce the configuration overhead is **to reduce the configuration data needed**. In this regard there is a big difference between fine-grained and coarse-grained DRHW. Fine-grained devices, such as FPGAs, need much more data because they are programmed at bit level. Each LUT and routing resource needs to be programmed, which require lots of bits in order to record such detailed configuration information. However, coarse-grained devices are programmed at word-level, which implies that they are programmed in a much simpler way with fewer control bits. Hence, another straightforward way to reduce the reconfiguration overhead is to simply use coarse-grained reconfigurable devices.

Nevertheless, this reduction on the complexity of the bitstreams comes at the price of losing flexibility. Hence only a limited amount of tasks can be targeted in a coarse-grained reconfigurable architecture, as long as the computations of these tasks can be specified in terms of the available hardware elements existing in the device (usually ALUs, registers, multipliers…). Many coarse-grained reconfigurable architectures have been proposed in the literature, most of which are purely restricted to the academic world. An interesting survey can be found in [Harte01].

3.3.2. Reducing the number of reconfigurations

In addition to reducing the size of the bitstreams, another way to reduce the reconfiguration overhead in reconfigurable systems is to perform fewer reconfigurations. There are several techniques developed in the literature to attain this objective.

One of them has been referred to as configuration caching [Li00, SNG01]. The basic idea is to prevent some reconfigurations from being replaced in the reconfiguration memory, following some criteria. Hence, this technique views the area of the FPGA as a cache. If this cache is large enough to hold more than one configuration, caching techniques are used to determine when the configurations should be loaded and unloaded to minimize the overall reconfiguration times. This is very useful in a practical scenario where it is likely that the same task will run several times on the device. In fact, if the caching technique is able to keep loaded in the memory those tasks which reconfigurations generate the greatest penalties in the execution of the system and/or the mostly frequently executed ones, the total reconfiguration overhead can be greatly reduced.

[Li00] is an interesting discussion about configuration caching techniques. In this work, authors propose different specific caching algorithms for each one of the following FPGA models: single-context, multi-context and partial run-time reconfigurable. In addition, for each FPGA model they propose three different types of algorithms: **run-time**, **complete prediction** and **general off-line algorithms**:

- **Run-time caching algorithms** only use information that is available at run time: the reconfigurations that have been performed up to the moment when the algorithm has to make a decision. Hence, these algorithms have to make guesses about the future behaviour of the program. This is similar to run-time cache management algorithms such as the Least Recently Used (LRU) policy. Due to the limited information that these algorithms handle, the decisions that are made may be wrong, which can even increase the total reconfiguration overhead.
- **Complete prediction caching algorithms** use the entire sequence of reconfigurations that will be performed in the future. Hence, these approaches provide an upper bound on reconfiguration overhead.

However, many times it is impossible to have this information at design time; hence in these contexts this upper bound is unachievable.

- **General off-line algorithms** represent an intermediate point between the two aforementioned kinds of algorithms. They work under the assumption that certain profiling information is available about the whole execution of the application, in addition to the information available at run time. Hence these approaches can make highly accurate predictions at run time. These algorithms typically perform between the run-time and the complete prediction ones in terms of quality, and they are realistic for most scenarios.

All the techniques proposed in [Li00] share the same idea: A design can be divided into a number of small blocks with a known execution sequence. Blocks are put together into groups, and each group can fit in the available hardware. A reconfiguration corresponds to the transition from one block to another when both blocks do not belong to the same group. Therefore, by optimally grouping the blocks, the number of required reconfigurations can be minimized and the total configuration latency is reduced.

Out of the three aforementioned kinds of algorithms, general off-line ones have been most extensively studied in the literature. An interesting related approach has been proposed in [SNG01], where the authors present two algorithms to cache tasks depending on the size of the reconfigurations and the order in which they have been used. Both techniques are referred to as **penalty-based** and **history-based algorithms**, respectively.

On the one hand, the **penalty-based algorithms** assign a cost to each reconfiguration that is present in the fabric, according to its size and the time that has passed since that reconfiguration was loaded in the FPGA. Thus, the smaller it is and/or the more time has passed since that reconfiguration was loaded, the greater this cost will be. When a new task comes in and there is no free space in the FPGA, they call a function to select the victim (or victims) to be replaced

according to their costs. The algorithm replaces the victim(s) with the highest cost. At first, it tries to replace just an existing task. However, if the selected victim does not free enough space, the function uses a two-slot window and tries to replace two adjacent reconfigurations which sum of costs is as much as possible. If it is not possible with a two-slot window, they try with a tree-slot one, and so on until it finds a feasible replacement.

On the other hand, the **history-based algorithms** try to predict which of the existing reconfigurations in the fabric will be used the latest, and tries to replace it. For this purpose, the algorithm must keep track of the reconfigurations that have been loaded and executed in the device so far. Next, they assign them a cost according to the following criteria: the less recent a task has been executed, the greater its cost will be. The rest of the process (i.e. the way the algorithm selects the victims) is similar to the penalty-based algorithm. Their results show that the history-based algorithm hides approximately 20% more latencies than the penalty-based one, although they are still far from the optimal results.

3.3.3. Scheduling techniques

Finally, a last way to reduce the reconfiguration overhead is to take it into account during the scheduling process. One effective technique that has been widely used in partially reconfigurable systems is task prefetch [Hauc98, LH02]. This technique consists in loading the reconfigurations in advance; i.e. before they are needed. Hence the reconfiguration times can be overlapped with the execution times of other previous tasks and hence the negative impact of the reconfiguration overhead can be reduced. Of course, this technique can only be applied if a reconfiguration of a task in a partial reconfigurable device does not have any negative effect over the existing tasks in the system. For instance, according to the vendors, in the latest Xilinx™ devices that support partial reconfiguration, the tasks already loaded in the system are guaranteed to be functional while a configuration process is being performed.

Task prefetch has been extensively used in many schedulers proposed in the literature. For instance, in [Hauc98], S. Hauck presents a scheduler that applies this technique in order to reduce the reconfiguration overhead. The target system used in this work comprises a general purpose processor and a reconfigurable coprocessor where hardware tasks are loaded. The prefetch is applied here by inserting special instructions in specific points of the code that runs in the processor. Thus, if the prefetch instruction is inserted far enough from the moment when the involved hardware operation is needed, its reconfiguration is successfully completed and its overhead is hidden. In addition, in [LH02] the authors extend the approach firstly introduced in [Hauc98]. Basically, they describe three different configuration prefetch algorithms: static, dynamic and hybrid. The main novelty here is the inclusion of a run-time technique to load reconfigurations in advance making predictions based on the recent history (dynamic prefetch) and to combine it with the static approach (hybrid prefetch).

The idea of task prefetch is also used in [QSN06]. In this work, the authors propose a new configuration model in order to perform several configurations simultaneously. For that purpose, they divide the configuration-SRAM of the FPGA into several individual tiles so that each one of them is accessed individually. This requires including as many reconfiguration controllers as tiles there are in the device to enable performing the maximum number of reconfigurations in parallel. In this context, the authors have developed a prefetch algorithm that is applied at design time and which main goal is to perform each reconfiguration as soon as possible using an available pair tile-controller. The main drawback of this algorithm is that, so far, commercial platforms include just one reconfiguration controller. Hence the idea of reconfiguration parallelism only exists from the theoretical point of view.

Task prefetch can be combined with other additional techniques, such as bitstream compression or reconfiguration caching, which have been mentioned above. This leads to more complex approaches that multiply the respective separate benefits of each applied technique. Thus, really elaborated task scheduling algorithms can be designed. This field has caught the attention of

many research groups in recent years. Hence, there are plenty of contributions in the literature regarding task scheduling. However, all of them can be classified into two big groups: **design-time scheduling** and **run-time scheduling**, depending on when the scheduler takes its decisions: at design time or at run time, respectively. **Hybrid scheduling** techniques have also been proposed. The rest of this subsection presents an overview of the contributions regarding task scheduling that are most related to the work presented in this PhD dissertation.

a) Design-time scheduling

Design-time scheduling consists in applying the scheduling techniques *before the application is executed*. In the literature, these approaches are also referred to as static approaches, since they are not able to deal with dynamic events that may happen at run time; i.e., during the execution of the tasks. However, their main advantage is that all the scheduling decisions are performed at design time; hence no additional overhead is generated at run time due to the scheduling computations. This problem has been extensively studied by a great number of research groups. Hence, this subsection only focuses on the most relevant related works.

An interesting work in this field has been presented in [GNS04]. It presents an optimal scheduling algorithm to execute directed acyclic graphs on a reconfigurable system that comprises a set of equal-sized reconfigurable tiles. The authors define a simplified model with the following assumptions:

- All the tasks that comprise a task graph are equal-sized.
- They assume that the execution times of the tasks are always zero. Hence, no prefetch can be applied in order to hide the reconfiguration overheads.

- All the task graphs are executed sequentially; i.e., only one task graph can be executed at a time.

Under these assumptions, the algorithm determines:

- **The sequence of reconfigurations** of the given task graphs in such a way that the number of reconfigurations that are carried out during their execution is minimal.
- **The replacement decisions** that must be carried out whenever a new task has to be loaded and there are no free reconfigurable slots. This problem, also known as the off-line paging problem, has been optimally solved by Belady [Bela66] for cache-based systems. This algorithm, called Longest Forward Distance, proves that the candidate that is going to be requested the furthest in the future is the best one to overwrite. Hence, this approach leads to the minimum number of page faults. There is a one-to-one correspondence between the problem presented in this work and the offline-paging problem. The Longest Forward Distance technique is known to be optimal for the latter; hence, it is also optimal for this problem.

The proposed approach in [GNS04] combines the Longest Forward Distance replacement strategy with a customized extension that specifies the order in which the reconfigurations of the incoming task graphs must take place. Under the aforementioned assumptions, the authors demonstrate that this approach achieves the optimal (i.e. minimum) reconfiguration overhead. Hence, it clearly outperforms other related works.

However, the main limitation of this work is their simplified environment, in which they assume that the execution time of the tasks is zero. Indeed, this unrealistic assumption prevents the algorithm from performing prefetch; which, as

commented in the previous subsection, is a powerful means to hide the reconfiguration latencies. On the contrary, these simplifications made possible to implement this algorithm in an embedded processor and use it in a real prototype, which is a tracking system that comprises a set of cameras and a controller that gathers the tracking information.

The second work that I want to comment about this topic is the research line followed by the group of M. D. Santambrogio et al. in [CRR+09, SRM09]. In these works, the authors provide a mathematical formalization for the problem of scheduling of a single task graph on reconfigurable devices, employing a very general model that does not skip any potential feature about the target platform or the nature of the incoming task graphs. Hence, they deal with task partitioning, scheduling and placement of the tasks in order to obtain the final solution. The algorithm has been built for the 1-D reconfigurable model.

In this context, the partitioning, mapping and the scheduling of the tasks are not independent. Hence they should be solved together in order to achieve optimality. However, even at design time this approach is unpractical, because of the overwhelming size of the search space, especially if the algorithm characterizes the tasks and the target system at such level of detail. Therefore they perform the partitioning and the mapping-scheduling separately. First of all, the partitioning phase receives the initial task graph and obtains from it a set of non-overlapping subgraphs which tasks and structure are as similar as possible. The idea is to promote task reuse. Then, for the mapping-scheduling phase, they use an external scheduler called Napoleon that applies an As Late As Possible (ALAP) approach to perform the task reconfigurations. This scheduler exploits configuration prefetch, module reuse and anti-fragmentation techniques.

Design-time scheduling can also be useful for applications with real-time constraints, provided that the features of the involved tasks and their periodicities are known in advance. Scheduling under real-time constraints on systems based on DRHW has also been studied in the literature. For instance, in [DMP06], the authors propose a real-time scheduling methodology and implement it in an

FPGA. In this work the authors characterize the tasks by means of their period, worst-case execution time and required amount of hardware resources. The objective of the algorithm is to find a feasible schedule in which all the tasks meet their deadlines, but also minimizing the resources consumption. For that purpose, they define several metrics to determine whether the solution is feasible or not. The proposed technique applies an "*Earliest Deadline First*" approach, but grouping iteratively several tasks in "contexts" to efficiently apply full reconfiguration. Each context represents a set of tasks that can run in parallel. Hence, as the algorithm merges several tasks in contexts, the obtained solution is closer to be feasible, but at the cost of increasing the resources consumption.

However, the main limitation of design-time scheduling techniques is that they can only obtain good results if they deal with applications which behavior is well-defined at design time. Hence, they are unsuitable for contexts with certain degree of dynamism.

b) Run-time scheduling

Run-time scheduling has also been studied in the literature. The main difference with respect to design-time scheduling is that in this case the decisions are made *while the application is being executed*. These approaches are also known as dynamic approaches, since they are able to deal with dynamic and unexpected events that are not known in advance; for instance, the arrival of a new task to be executed. However, their main drawback is that the scheduling computations must be performed at run time. Hence in this case the scheduling process should not be computationally intense in order to avoid generating too many additional temporal overheads.

The group of J. Noguera and R. M. Badía has been very active in the field of task scheduling. In their previous works, they propose a codesign methodology [NB02a, NB01] with the goal of obtaining the task representation (i.e.

dependences between tasks), estimate their timing and area requirements, and performing the hardware/software partitioning in order to decide which tasks will be executed in DRHW and which ones in software. Once this previous flow has been completed, the run-time scheduling has to be performed. In this regard, two interesting contributions have been presented in [NB02b, NB04].

In [NB02b] the authors present a run-time scheduler for DAGs to be targeted in a reconfigurable architecture divided into a set of Reconfigurable Units (RUs) of equal size. The nodes of the DAGs have also a priority assigned. The proposed scheduler follows an ASAP event-based approach, in which specific events are generated in certain moments of time; i.e. whenever a task can be reconfigured or executed. When an event is generated, it is captured and the proper scheduling actions are carried out, which basically consist in loading and executing the tasks in the reconfigurable device as soon as possible. Hence, a prefetch technique is implicitly applied here in order to reduce the impact of the reconfigurations. This scheduling process is performed taking into account the data dependencies among the tasks and their priorities. This means that a task cannot start its execution until all its predecessors have already been executed. In addition, if several tasks can be loaded at a given scheduling step, the task with the greatest priority is scheduled first. In [NB04] the authors extend this work by proposing a hardware implementation of such a scheduler. This architecture includes two modules that run concurrently and interact using a shared memory. There is a producer process (named *Graph Dependence Check Logic*) and a consumer one (named *Dynamic Scheduling Algorithm*). Both processes produce and consume the events, which are stored in the shared memory.

However, in these two works it is assumed that the target architecture is able to perform several reconfigurations in parallel. This is not a realistic assumption, since nowadays no commercial reconfigurable architecture enables configuration parallelism.

The group of H. Walder and M. Platzner has also presented interesting contributions in the field of run-time task scheduling. For instance, in [WP03] they

present an on-line scheduler for hardware tasks in a reconfigurable device that is partitioned into a set of reconfigurable blocks with different sizes [WP04]. (Section 3.1). The scheduler groups the tasks that are waiting for their execution into several queues. These queues can be assigned to one or several reconfigurable blocks, with the restriction that the tasks belonging to a given queue cannot be bigger than any of the blocks assigned to that queue. As in the work presented in [NB02b, NB04], this run-time scheduling approach is also event-based. In this case, the load and execution of a task is triggered either by the end of the execution or the end of the reconfiguration of a task; or by the arrival of a task with a higher priority.

In this work the authors present several versions of the same scheduler. On the one hand, the scheduler can work in either of the two following operative modes: *restrict mode* and *prefer mode*. When the system operates in *restrict mode* it applies a *best fit* policy, assigning each task to one of the smallest blocks where the task fits, whereas in *prefer mode* the system may assign the task to any block as long as it fits. On the other hand, the scheduler can implement any of the following four replacement strategies: First-In First-Out (FIFO), Shortest Job First (which are non-preemptive), Shortest Remaining Processing Time and Earliest Deadline First (which are preemptive). Preemptive strategies can abort the execution or the reconfiguration of a task when a task with a higher priority arrives. This makes the scheduler more flexible. However, it takes a long time to pause and resume a reconfiguration or execution of a task, which can be unaffordable in some contexts.

c) Hybrid design-time/run-time scheduling

The last two subsections introduced the concepts of design-time and run-time scheduling. Both of them differ in the moment the scheduling is applied: before the execution of the applications or while their execution is taking place, respectively. Each one has its advantages and drawbacks. On the one hand, in

design-time scheduling all the decisions are already made when the application starts its execution. Hence, no computations are performed at run time and the scheduling process does not generate any additional temporal overhead due to these computations. This allows the scheduling technique being as complex as necessary in order to obtain optimal or near-optimal schedules. However, these approaches can only be applied when very limited dynamic behavior exists. On the other hand, run-time scheduling assumes that all the decisions are made while the application is running. This makes possible to deal with very dynamic scenarios, for instance when the sequence of tasks that are going to be executed in the future is totally or partially unknown in advance. However, a run-time scheduling algorithm should not be too complex since carrying out too many computations at run time can significantly degrade the performance of the running applications. This may limit the efficiency of the run-time scheduling technique, especially when a complex approach is needed in order to obtain good schedules.

Hybrid design-time/run-time scheduling is intended to be something in the middle between these two ends. The basic idea is to apply a run-time approach to deal with certain degree of dynamism, but carrying out as many computations as possible at design time in order to save run-time computations. Such scheduling hybrid techniques have also been proposed in the literature. For instance, two interesting related approaches have been proposed in [RVM+04] and [RMC05].

In [RVM+04] the authors propose a hybrid design-time/run-time scheduling flow for tasks in a hardware reconfigurable architecture. This flow is depicted in Figure 3.4. The main idea is to generate several schedules at design time and then, to select at run time the most appropriate one, depending on the dynamic status of the system. In addition, this scheduler also carries out some run-time optimizations, performed by the *reuse* and *prefetch* modules.

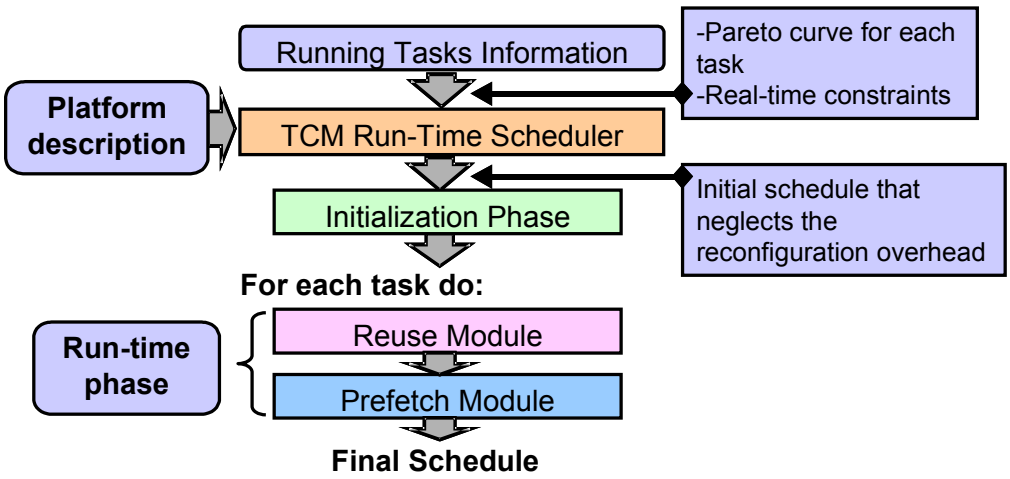

Figure 3.4. Hybrid scheduling flow proposed in [RVM+04]

The initial schedules are obtained through the *Task Concurrency Management* (TCM) scheduling environment [YWM+01, YC03]. In TCM an application is represented as a set of tasks, and each task is described as a DAG. Instead of generating a sole solution, for each task the TCM scheduler generates a Pareto curve [EKO90], which represents a set of solutions (or Pareto points) where each solution is better than all the others in at least one of the parameters to optimize. In this work these parameters are execution time and energy consumption. Hence, in this context the Pareto curve represents a set of pseudo-optimal schedules, in such a way that a Pareto point is not better than another one in both parameters. Once this set of possible schedules has been generated, the run-time scheduler selects the solution that better adapts to the dynamic conditions of the system. For instance, if the execution of a given task is urgent, the scheduler selects a quick schedule for it, but at the cost of consuming much energy. Otherwise, the scheduler can select a slower one, thereby reducing the energy consumption.

Since TCM schedulers do not take into account the reconfiguration overhead of the DRHW resources, the algorithm presented in [RVM+04] adds this information to the selected solution. This is performed in the "*Initialization Phase*" (Figure 3.4). Finally, the run-time phase tries to reduce this overhead by applying

some optimization techniques. First of all, the "*Reuse Module*" tries to maximize the amount of tasks reused. The objective of this phase is to avoid performing the reconfigurations of some tasks by reusing them, provided that they are already loaded in the system as a result of a previous execution. Then, the "*Prefetch Module*" tries to hide the rest of the reconfigurations by applying a prefetch technique similar to the one presented in [Hauc98].

In [RMC05] the authors propose a hybrid design-time/run-time prefetch approach and integrate it in the scheduler initially proposed in [RVM+04]. This technique was already described in detail in Chapter 2, and is actually one of the techniques that were evaluated experimentally in Section 2.2. Hence this section will not describe it again.

However, in spite of these two interesting contributions, the field of hybrid task scheduling has to be more widely explored. In contrast with fully design-time and run-time scheduling, no many more contributions can be found in the literature about hybrid techniques. In addition, the existing approaches lack more elaborated scheduling support to deal with the execution of several consecutive tasks under highly dynamic conditions. Furthermore, to the best of our knowledge, no current work presents a hardware implementation of such a scheduler and evaluates its performance.

The work presented in this PhD dissertation can be classified in the field of hybrid design-time/run-time scheduling. I consider that this work presents important breakthroughs with respect to other state-of-the-art approaches. On the one hand, the scheduler developed in this doctoral thesis does not take as input the results from an external design-time scheduler (such as [RVM+04] and [RMC05] do with TCM) and make some modifications on it at run time, but it makes its own scheduling decisions. Moreover, we have developed several new techniques that allow applying important improvements, such as an intelligent reuse heuristic and a technique to avoid falling into optimal local solutions, as Chapters 4 and 5 describe in greater detail. On the other hand, this is the first

time that a scheduler for DAGs execution in reconfigurable multi-tasking systems have been implemented in reconfigurable hardware and evaluated.

3.4. Conclusions

DRHW has been gaining popularity both from the academic and commercial point of view in the last years. In fact, there are more and more research groups making significant efforts to improve the usability of reconfigurable devices. In addition, nowadays we are witnessing the emergence of new companies proposing new reconfigurable architectures, whereas the most consolidated ones (such as Xilinx™) have been considerably increasing their sales lately.

The best example of this may be that many SONY products, such as Network, CD and HI-MD Walkman™ and PSP™, include a reconfigurable co-processor called Virtual Mobile Engine (VME™)[10]. SONY has been the first manufacturer ever to include a module based in DRHW technology in the multimedia market (their Network Walkman NW-MS70D, released in February, 2003). The success of this platform has come thanks to the higher performance that the VME™ co-processor provides and to its lower energy consumption (between four and five times less than their general purpose processors). This is a perfect example of the potential benefits of including DRHW in commercial architectures. Hence it is reasonable to think that DRHW may become a widespread technology in the years to come, and not only restricted to the academic world.

However, in spite of these attempts to make room for DRHW in the semiconductor industry, reconfigurable hardware is still far from being a serious alternative to commercial general purpose processors at least in the near future. The main reason of this is that sometimes it is extremely difficult to take the most advantage of the features of the reconfigurable devices, mainly due to the lack of support that the manufacturers offer nowadays.

This implies that in many cases, the designer has to face directly many low-level issues related to DRHW. One of the most important and studied ones is the

[10] www.sony.net/Products/SC-HP/cx_news/vol42/pdf/sideview42.pdf

reconfiguration overheads that a reconfiguration process involves. This is the reason much related work has already been proposed at the academic level to tackle this problem. As overviewed throughout this chapter, these contributions can be classified in: a) reducing the size of the reconfiguration bitstreams, b) reducing the number of reconfigurations and c) taking the configuration process into account during the task scheduling. In particular, the work presented in this PhD dissertation falls into the latter two groups.

The field of task scheduling has been object of study by many research groups in the literature. These contributions can be classified either as design-time or run-time schedulers, although hybrid techniques have also been proposed. The latter is a very interesting field of study because such techniques take the best of both fully design-time and run-time approaches. On the one hand, the bulk of the computations are performed before the execution of the application, as long as the needed data are available. Since these computations do not generate any run-time latency, very complex computations can be performed in this phase in order to obtain optimal or near-optimal schedules. On the other hand, some additional computations are performed at run time depending on the dynamic status of the system. This makes the scheduler able to deal with unpredictable run-time events. Hence a hybrid approach is certainly the best scheduling choice in many contexts.

Many contributions can be found in the literature proposing scheduling techniques that are applied either fully at design time or fully at run time. However, not so much work has been published about hybrid design-time/run-time scheduling. The work developed in this doctoral thesis has as objective to present a contribution in this yet unexplored field. Hence, I consider that this research line is totally innovative and clearly relevant to the scientific community, and that this work, in addition to other related works about this topic, will promote future research in the field of DRHW.

Genius is one per cent inspiration,
ninety-nine per cent perspiration

T. A. Edison

Chapter 4:
The proposed scheduling algorithm

As hinted in the past chapter, the scheduler that has been developed in this doctoral thesis is divided into a design-time phase and a run-time one. The idea is to perform an important part of the calculations at design time in order to reduce the run-time penalty generated throughout the scheduling process. However, not all the computations can be performed at design time, since in order to obtain good results the scheduler must take into account the current status of the FPGA. This is the reason some computations must still be performed at run time.

This chapter describes in detail the proposed scheduling flow. Section 4.1 describes the design-time phase, which objective is to obtain some information about the task graphs to be executed that will be used later at run time. Next, Section 4.2 describes the run-time phase, which applies prefetch and

replacement techniques in order to hide the reconfiguration overhead generated during the execution of the tasks. Finally, Section 4.3 summarizes this chapter with some final conclusions.

4.1. The proposed scheduling algorithm: design-time phase

The objective of this phase is to extract useful information about the incoming task graphs that will be used at run time to make good scheduling decisions without carrying out complex computations. Basically, this phase characterizes each task with three parameters: **weight**, **criticality** and **mobility**, as the flowchart of Figure 4.1 shows. The next three subsections describe these steps in greater detail.

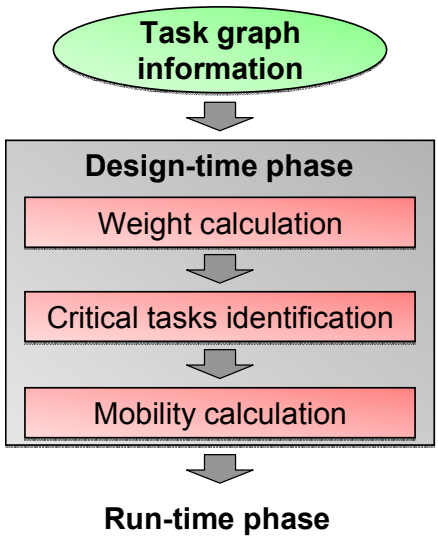

Figure 4.1. Flowchart of the design-time phase of the proposed scheduling algorithm

4.1.1. Weight calculation

The input data that the scheduler receives is just the specification of the task graphs to be executed. Each task graph comprises the set of nodes (each node is a computational task) and the data dependencies existing among them. We assume that this information also includes reliable estimations of the execution

time of each task. Nevertheless, if the execution time of the tasks was variable (for instance, depending on the input data), the scheduler could adopt a solution similar to the one proposed in [WMY01], where the authors suggest creating several versions for the same task graph with different execution times (called *scenarios*) and then choosing the one that best fits to the current conditions. These scenarios are identified at design time; hence if it was necessary to apply this technique, our scheduling flow would continue being valid.

With this information, the first calculation that the scheduler performs is to obtain the order in which these tasks will be reconfigured. This is of critical importance for the performance of the system, since the sooner a task is reconfigured in the system, the earlier its execution can take place. However, as the target reconfigurable device only has one reconfiguration circuitry, these reconfigurations must be performed sequentially. Hence, the reconfiguration sequence must be decided according to some criteria. More specifically, we can say that for the sake of the performance, this sequence of reconfigurations must fulfil the following two conditions:

- A given task *t* should be loaded in the system before its successors, since the dependencies existing between *t* and its successors indicate that *t* will be needed earlier.
- Given two tasks without direct data dependencies between them, if one of them belongs to the critical path of the graph, then its reconfiguration must take place before. In fact, if that reconfiguration is delayed, then all the following tasks that belong to the critical path are delayed as well, whereas the reconfiguration of those tasks that do not belong to the critical path can be delayed (to a certain extent) without increasing the overall execution time of the system.

In order to determine this sequence of reconfigurations, the scheduler firstly assigns a weight to each task according to the criteria that is shown in (1):

$$(1) \quad W(t_i) = execution\_time(t_i) + MAX\{W(t_j)\}, \forall t_i \in TASKS, \forall t_j \in SUCCESSORS(t_i)$$

This formula means that the weight of a leaf task (which has no successors) is just its execution time; and for the rest of the tasks, their weight is the addition of their execution time and the maximum of the weights of all their successors. The formula covers both cases, since the successors of a leaf task is the empty set.

An example of this process is shown in Figure 4.2.a. Thus, according to the figure, *Weight*(Task 4)=6, which is its execution time, since Task 4 is a leaf task. Then, following the criteria depicted in (1), *Weight*(Task 2)=8+*max*{6}=14, *Weight*(Task 3)=12+max{6}=18 and *Weight*(Task 1)= 6+*max*{14, 18}=24.

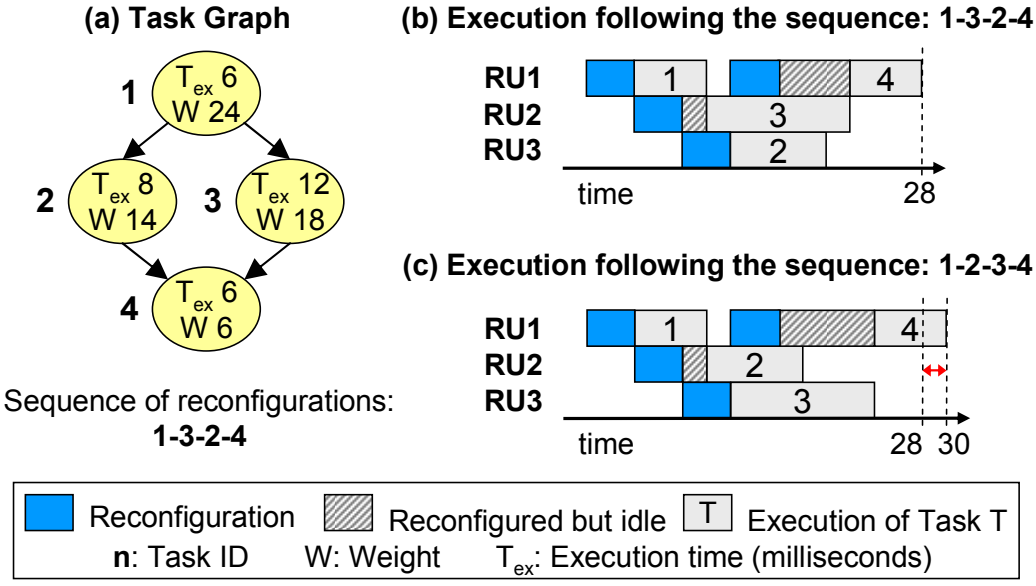

Figure 4.2. a) Example of the weight calculation algorithm. b) Execution of the example task graph following the obtained sequence of reconfigurations (1-3-2-4). c) Execution of the example graph following a sub-optimal sequence of reconfigurations (1-2-3-4). The reconfiguration latency is 4 milliseconds

The weights represent the impact of the tasks in the critical path of the task graph. Thus, if *Weight(*A*)* > *Weight(*B*)*, it means that Task A must be loaded before Task B. Hence, once this process is performed, all the tasks are sorted decreasingly according to their weights to determine the sequence of reconfigurations that the scheduler will follow at run time. Thus, according to Figure 4.2.a, this sequence is 1-3-2-4.

In order to illustrate the importance of determining an order in the reconfigurations, Figure 4.2.b and Figure 4.2.c show an example of execution of the same task graph following the sequence of reconfigurations obtained in Figure 4.2.a and compare it with a different sequence (1-2-3-4), which is obtained by just swapping the reconfigurations 2 and 3 from the optimal one. As the figure shows, the reconfigurations are performed sequentially as soon as possible and, if possible, in parallel with the execution of other tasks (prefetch). Section 4.2.2 of this chapter will explain in detail how this process is performed and how the scheduler decides in which RU to execute the tasks.

Figure 4.2.b shows that if the system follows the optimal sequence of reconfigurations, the load of Task 1 is the only one that generates any reconfiguration overhead. However, Figure 4.2.c shows what happens if the scheduler follows a different sequence. In this case, Task 3 (which belongs to the critical path) is loaded after Task 2 (which does not). And as the figure shows, an additional overhead of two milliseconds is generated.

4.1.2. Critical tasks identification

Once the sequence of reconfigurations has been obtained, the design-time scheduler identifies which tasks are especially critical for the system, in other words, those tasks which reconfiguration is especially negative from the point of view of the performance of the system.

As previously shown in Figure 4.2.b, the load of Task 1 generated a reconfiguration overhead at the beginning of the execution of the task graph. Independently of how its execution is scheduled, Task 1 always generates the same penalty, since this is the first task in the reconfiguration sequence and it cannot be prefetched with any other task in the graph. Hence, the only way to eliminate the reconfiguration overhead of Task 1 is by reusing it. In this case, we say that *Task 1 is a critical task* and it is the only critical one among the four tasks that the task graph of the example contains.

The objective of this second scheduling step is to obtain the set of these critical tasks. We define it as follows:

The critical task set is the minimum set of tasks that fulfill the following condition: If they are reused (and therefore they do not generate any delay due to their reconfiguration latency), the scheduler will be able to hide the reconfiguration overhead of the remaining (i.e. non-critical) tasks.

When, in this definition, we mention “the scheduler”, we refer to the run-time scheduler that will be described in the end of this chapter. Hence, the obtaining of the critical tasks involves using this run-time scheduler in advance; (i.e., before the actual execution of the task graph) in order to know whether the execution of a given task will generate any reconfiguration overhead or not. Therefore, this process can be seen as a previous simulation of the task graph. We can use our run-time scheduler in order to carry out these simulations at design time, since we already know how it will work at run time. However, since it takes into account dynamic information that is only known at run time, in order to perform this previous simulation, some dynamic variables need to be set in advance. More specifically:

- The run-time scheduler takes into account the criticality and mobility of the tasks throughout the scheduling process. This information is unknown at design time; hence for this previous simulation we assume that initially all the tasks are not critical and their mobility is 0.
- In addition, the scheduler also takes into account the information about which tasks are loaded in which reconfigurable units each time a replacement has to be performed, in order to promote the task reuse. Since this information is still undefined at design time, for these simulations we assume that the execution of the task graph is performed in a system with initially empty reconfigurable units (with no tasks loaded from a previous execution).

Hence, the critical task set obtained throughout this process not only depends on the features of the task graph, but also on the scheduler used to simulate its execution (in this case, our run-time scheduler). The reason is that it relies on it to obtain previous scheduling simulations that will be analyzed subsequently for the critical tasks identification process, as it will be explained in detail below. A good example is shown in Figure 4.2.c, which is a different schedule from the one shown in Figure 4.2.b. As the figure shows, in this case not only the reconfiguration of Task 1 generates an overhead, but also the one of Task 3, which is actually the additional latency that is introduced in schedule of Figure 4.2.c with respect to the one of Figure 4.2.b. Hence, if the second schedule is selected, *both Task 1 and Task 3 are critical*.

This means that any scheduler can be used at this point in order to perform these preliminary simulations, as long as it is the same scheduler that will be used at run time.

Once the scheduler has identified a task as critical, it also assigns it a value of criticality that represents the delay that it generates if it is not reused. At run time, the replacement policy will take into account this information assigning greater priority to the critical tasks.

Figure 4.3 shows the pseudo-code of this algorithm. Firstly, the set of critical tasks is initialized to the whole set of tasks in the task graph (Line 1). Then, the function *schedule (task_graph, CT)* (Line 2) is called. This function invokes our run-time scheduler (as explained above) assuming that all the tasks in the critical tasks set (*CT*) are reused and returns a reference schedule (*ref_sch*). Hence, as initially all the tasks have been assigned to the critical tasks set, in this step we obtain an *ideal schedule* with no reconfiguration overhead. This *ideal schedule* is used as a reference during the critical task identification process.

Critical tasks (CT) identification:

```
 1. CT := whole_set_of_tasks (task_graph);
 2. ref_sch := schedule (task_graph, CT);
 3. CT := Ø;
 4. end := 0;
 5. WHILE (not end){
 6.   current_sch := schedule (task_graph, CT);
 7.   IF (ex_time (current_sch) == ex_time (ref_sch)){
 8.     end := 1;
 9.   }ELSE{
10.    delayed_tasks := compare (current_sch, ref_sch);
11.    t := max_weight (delayed_tasks);
12.    add_critical_task (t, CT);
13.  }
14.}
15.RETURN CT;
```

Figure 4.3. Algorithm to identify the critical tasks

The objective of this process is to find a schedule that provides the same performance as *ref_sch*, but with the minimum number of tasks assigned to the critical tasks set. To start the search, we initialize this set as empty (Line 3) and we start an iterative process (Lines 5-14). Then, the *while* loop starts computing a new schedule (*current_sch*, Line 6) and compares its execution time with the execution time of *ref_sch* (Line 7). If both schedules have the same execution time, the iterative process finishes. Otherwise the *compare* function identifies which tasks have been delayed in *current_sch* (Line 10) with respect to *ref_sch*. In the next step (Line 11) the function *max_weight* identifies the task with the greatest weight among all the delayed ones in *current_sch*, and finally this task is

added to the critical tasks set (Line 12). The *while* loop continues iterating until the execution time of the current schedule is the same as the ideal one (Line 8). Finally, this function returns the *CT* set (Line 15).

Figure 4.4 depicts an example of this process. First of all, the algorithm invokes the run-time scheduler in order to obtain the reference schedule shown in Figure 4.4.b, assuming that all the tasks in the graph are reused. As mentioned above, this reference schedule is an upper bound in terms of performance for the final one. This reference schedule will be compared with the ones that will be obtained in successive iterations of the loop. Note that, as in the previous example of Figure 4.2, the scheduler prefetches the reconfigurations of the tasks by loading them as soon as possible in the system. In this case, the sequence of reconfigurations is exactly the same as in the example of Figure 4.2.a: 1-3-2-4.

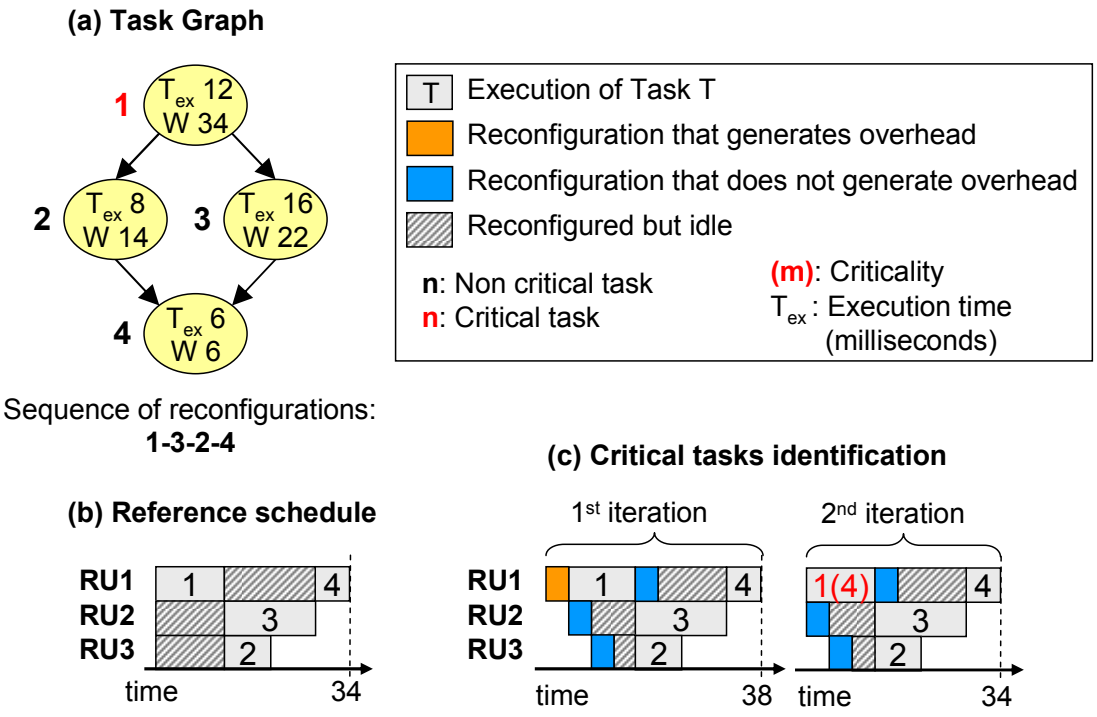

Figure 4.4. Example of the critical task identification. The reconfiguration latency is 4 milliseconds

Then, the algorithm labels all the tasks as non-critical and carries out the first iteration of the loop (Figure 4.4.c, *1st iteration*). Thus, it invokes again the run-time scheduler in order to obtain the schedule corresponding to this iteration and compares it with the reference one. It identifies a delay of 4 milliseconds compared to the ideal execution, and identifies that Task 1 is the only one that is delayed. Hence, this task is selected, labeled with a criticality of 4 and added to the critical tasks set.

In the following iteration of the loop (Figure 4.4.c, *2nd iteration*), the scheduler is invoked again and assumes that Task 1 is reused. In this case the configurations of Tasks 2, 3 and 4 do not generate any execution time overhead. Hence the algorithm stops and Tasks 2, 3 and 4 remain labeled as non-critical.

4.1.3. Mobility calculation

In the previous example of Figure 4.4, the run-time scheduler was invoked several times in such a way that the reconfigurations were performed ASAP. However, the scheduler could also delay some of them and still obtain an optimal schedule regarding performance. Hence, in this case it does not follow a purely ASAP approach. The objective of this last step is to identify the *mobility* of the tasks; in other words, to what extent their reconfiguration can be delayed without generating any additional reconfiguration overhead. This gives more flexibility to the final schedule and helps to avoid falling into sub-optimal solutions, especially in the execution of several task graphs sequentially, as it will be shown next.

Figure 4.5 shows a motivational example of this idea and illustrates how our scheduler can escape from local-optimum solutions by delaying some reconfigurations. The task graph of the figure is exactly the same as in the previous one and it is executed twice in a system with three reconfigurable units. Its sequence of reconfigurations is again the same: 1-3-2-4. The figure depicts

the execution obtained when applying prefetch with an ASAP approach (b) and when our scheduler delays some reconfigurations (c).

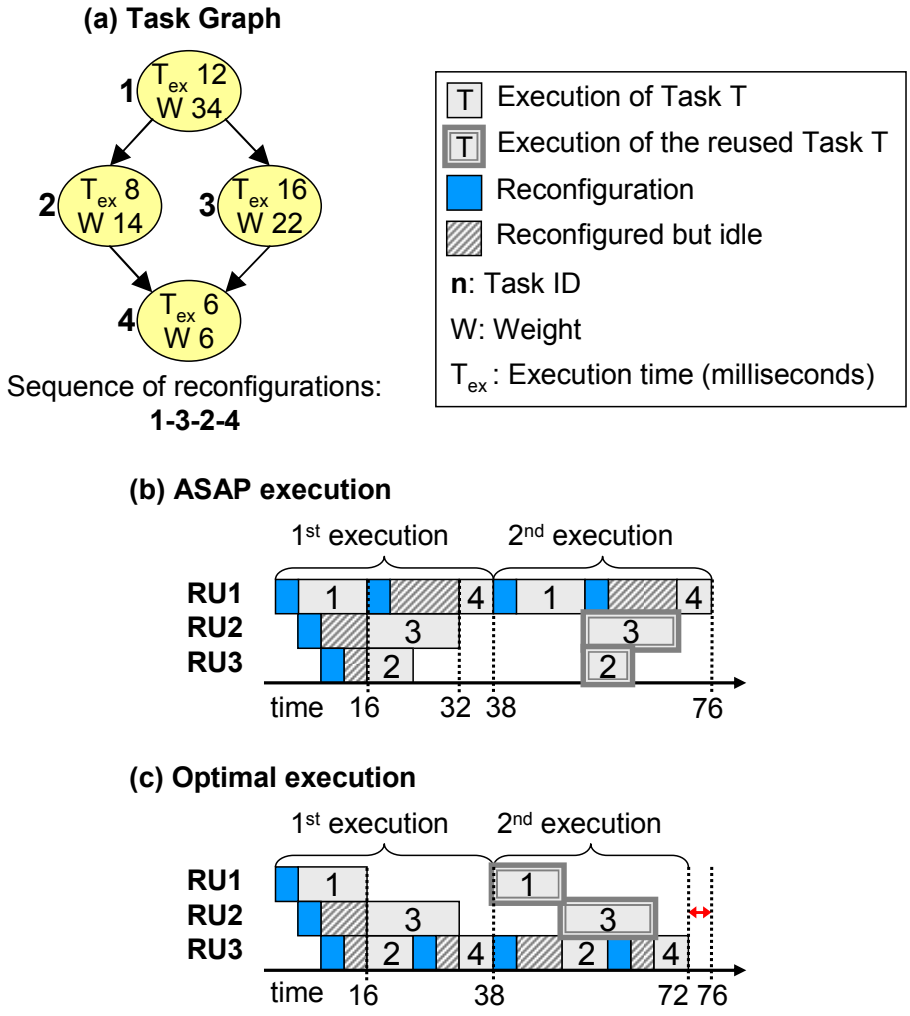

Figure 4.5. Execution of a task graph (a) in a platform with three RUs applying an ASAP approach (b) and using our scheduling technique that achieves the optimal solution delaying one of the reconfigurations (c). The reconfiguration latency is 4 milliseconds

In both approaches we can observe that the prefetch technique is very powerful when dealing with DAGs since it hides the latency of most reconfigurations. The simplest way to apply prefetch is using an ASAP approach, since it is very simple to implement and it generally leads to good schedules. For instance, according to Figure 4.5.b, with this approach the latency of three of the

four tasks is hidden and Task 1 is the only one that generates delays due to its reconfigurations. However, during the scheduling process, Task 1 is replaced by Task 4 immediately after its execution; since it tries to load it as soon as possible and in that instant, the reconfigurable unit 1 (where Task 1 is loaded) is the only available candidate. For this reason in the second execution of the same task graph, Task 1 has to be loaded again and its reconfiguration introduces another delay in the execution.

This overhead disappears if we delay the reconfiguration of Task 4 (Figure 4.5.c). In this case, we assume that our scheduler knows that the reconfiguration of Task 4 can be delayed without any performance degradation. Hence, when Task 1 is selected as the replacement victim (*time* = 16 milliseconds), the scheduler decides to delay this reconfiguration waiting for the end of the execution of Task 2. Thus, when Task 2 finishes its execution, the replacement policy can select between two possible candidates (Tasks 1 and 2 in reconfigurable units 1 and 3, respectively), and it selects RU3 since Task 2 is not critical. As a consequence, when the same task graph is executed again, Task 1 is directly reused and no reconfiguration generates any delay in the execution.

Every time the scheduler has to carry out a reconfiguration or an execution of a task, it has to decide in which instants of time to carry them out. For instance, in Figure 4.5.b the reconfiguration of Task 4 is performed immediately after the execution of Task 3 (when *time* = 16 milliseconds); i.e., following an ASAP approach. However, this reconfiguration can also be scheduled at *any instant of time* in the middle between the ASAP and ALAP approaches ($16 \leq time \leq 28$ milliseconds, since a reconfiguration takes 4 milliseconds). Since there are many possible choices the scheduler can select, in order to simplify the decision space, the scheduler only considers *some discretized time values when the reconfigurations and executions of tasks can take place*. In other words, the scheduler makes decisions *only when certain events happen*, which in this particular case are generated whenever a task has finished its reconfiguration or execution. Section 4.2 will provide further details regarding the management of these events.

In any case, if the scheduler follows this event-based technique and applies an ASAP approach, it tries to load a given task as soon as an event has been generated, the reconfiguration circuitry is free and there is at least one available reconfigurable unit. However, if the scheduler decides to delay the reconfiguration of a given task, this means that it chooses to *skip a certain number of events* until it decides that it is a good moment to load that task. Therefore, the mobility of a task can be measured in terms of how many events can be skipped before reconfiguring a given task without generating any additional reconfiguration overhead.

The reason to delay a task is always the same: the replacement policy has selected as a victim a task that the scheduler prefers not to replace. Hence the scheduler checks the mobility, and if there is still margin for delaying the reconfiguration, it waits until the following event hoping that at that time the replacement policy will find a less important victim.

The objective of this third step is to assign a value of mobility to each task, in terms of how many events the scheduler can safely skip before its reconfiguration without generating any additional delay. Figure 4.6 shows the pseudo-code of this algorithm. Initially the mobility of all the tasks is initialized to 0. Then the process starts identifying the critical and the non-critical tasks of the task graph and stores them in CT and NCT, respectively (Lines 1-2). The critical tasks have no mobility because even if their reconfiguration is scheduled ASAP they will generate a delay, hence the following main loop (Lines 3-15) just iterates over the non-critical ones. In this loop, the algorithm sequentially extracts a task *t* from NCT (Line 4), and the function *schedule ()* invokes the run-time scheduler to obtain a first schedule (*ref_sch*) assuming that *t* has mobility 0. Then, the *do-while* loop (Lines 6-13) tentatively increases the mobility value of *t* (Line 7) and calculates a new schedule (*new_sch*), but this time delaying the reconfiguration of *t* as many times as the value of *t.mobility*. This is again done in the function *schedule ()* (Line 8). Then, the algorithm checks if it is feasible to assign that new mobility to *t* without degrading the performance of the system (Line 9). Thus, if *diff*>0 (Line 10), an additional overhead has been generated; hence the algorithm

restores the former mobility value to *t* (Line 11) and exits from the *do-while* loop (Line 13). Otherwise, the mobility of the task is tentatively increased again and the *do-while* loop carries out another iteration. When this loop finishes, the involved task is removed from NCT. The *while* loop continues iterating until NCT is empty.

```
Mobility assignment:
 1. CT := obtain_critical_tasks (task_graph);
 2. NCT := obtain_non_critical_tasks (task_graph);
 3. WHILE (NCT ≠ Ø){
      /* Get a task from NCT */
 4.   t := get_task (NCT);
      /* Schedule with previous t.mobility */
 5.   ref_sch := schedule (task_graph, CT);
 6.   DO{
 7.     t.mobility++;
        /* Schedule with new t.mobility */
 8.     new_sch := schedule (task_graph, CT);
 9.     diff := ex_time (new_sch) – ex_time (ref_sch);
10.     IF (diff > 0){
11.       t.mobility--;
12.     }
13.  }WHILE (diff == 0);
14.  remove (t, NCT);
15.}
```

Figure 4.6. Algorithm to assign the mobility to the non-critical tasks

Figure 4.7 illustrates this process with an example. Firstly, the algorithm selects the non-critical tasks of the graph: 2, 3 and 4. For Task 3 (Figure 4.7.c), it attempts to assign a mobility of 1. However, when the scheduler delays its reconfiguration, it generates a delay in the execution of 4 milliseconds. Hence, the mobility of Task 3 is set to 0. On the contrary, the reconfiguration of Task 2 (Figure 4.7.b) can be delayed once without any performance degradation. Hence the scheduler assigns them a mobility of 1 in the first iteration. Afterwards it attempts to assign a mobility of 2, but in this case a new overhead of 4 milliseconds is generated. Hence, the mobility of Task 2 is set to 1. The same process is repeated for Task 4 (Figure 4.7.d), which mobility is also set to 1.

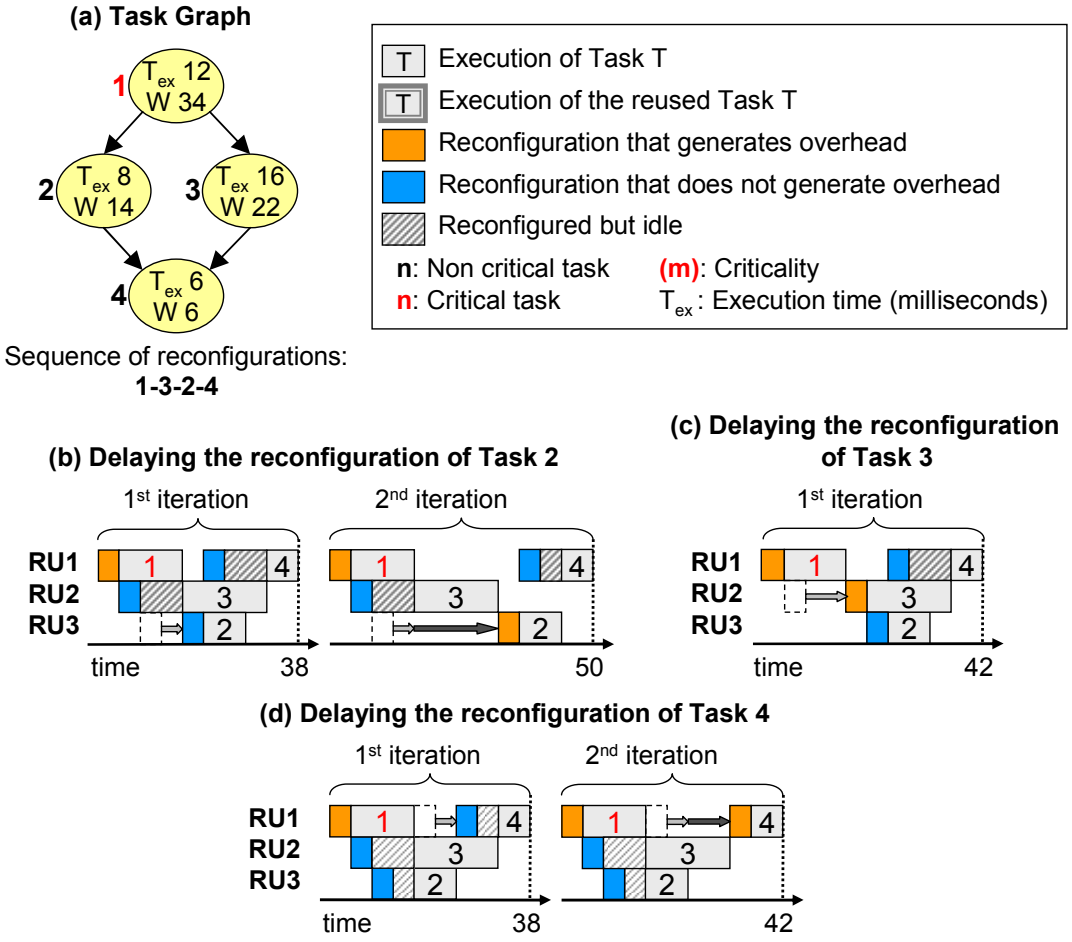

Figure 4.7. Example of the mobility calculation. The reconfiguration latency is 4 milliseconds

4.2. The proposed scheduling algorithm: run-time phase

At run time the scheduler steers the execution of the task graphs taking into account their internal dependencies, the information obtained at design time and the dynamic status of the system. For this purpose, in this phase the scheduler exploits task prefetch and reuse in order to reduce the reconfiguration overhead during the execution of the task graphs.

As shown in Figure 4.8, the run-time scheduler comprises two modules that collaborate between them and perform the *task-graph execution management* and the *replacement technique*, respectively. The next two subsections explain their functionality in greater detail.

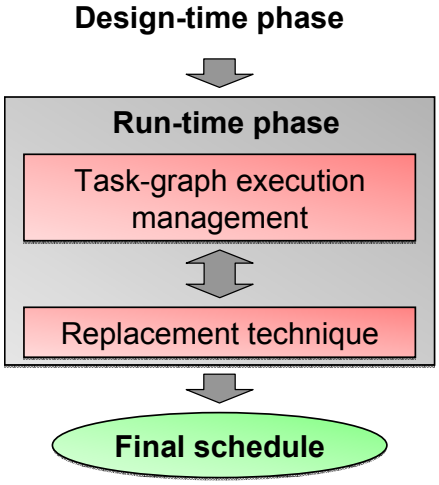

Figure 4.8. Flowchart of the run-time phase of the proposed scheduling algorithm

4.2.1. The task-graph execution manager

Basically, this module guarantees the correct execution of the task graphs taking into account their data dependencies and the dynamic status of the system. As hinted above in the Section 4.1.3, the scheduler only considers some discretized time values following an event-triggered approach. This means that the scheduler only makes decisions when certain important events happen. This approach greatly reduces the complexity of the run-time scheduling process, but at the same time it provides enough flexibility to optimize the execution. The manager supports four types of events:

- ***new_task_graph***: this event is generated when a new task graph has to be executed.
- ***end_of_execution***: this event is generated each time the execution of a task has finished.
- ***end_of_reconfiguration***: this event is generated each time the reconfiguration of a task has finished.
- ***reused_task***: this event is generated when a task is reused. This happens when the scheduler tries to load a task and identifies that it is already loaded in the system.

The *end_of_reconfiguration* and the *reused_task* events are similar from the point of view of the manager, since reusing a task is equivalent as carrying out a reconfiguration in zero clock cycles. The task-graph execution manager sequentially processes the events generated in the system and carries out the proper scheduling actions. Figure 4.9 depicts the actions triggered by each event.

When a new task graph arrives (Line 1) and the reconfiguration circuitry is idle (Line 2), the system attempts to schedule the reconfiguration of the first task in

the sequence of reconfigurations of the received task graph, which was obtained at design time (Line 3).

```
/* task = task that triggered the event
 * RC = Reconfiguration Circuitry */
CASE event IS:
 1. new_task_graph:
 2.     IF (RC == idle){
 3.         look_for_reconfiguration (&rec_sequence);
 4.     }
 5. end_of_reconfiguration or reused_task:
 6.     check_dependencies (&task);
 7.     IF (is_ready (&task)){
 8.         start_execution (task);
 9.     }
10.     look_for_reconfiguration (&rec_sequence);
11. end_of_execution:
12.     IF (RC == idle){
13.         look_for_reconfiguration (&rec_sequence);
14.     }
15.     update_task_dependencies (&task);
16.     FOR (i := 0 TO NUMBER_OF_RUS){
17.         check_dependencies (&task);
18.         IF (is_ready (&task)){
19.             start_execution (task);
20.         }
21.     }
```

Figure 4.9. Pseudo-code of the run-time task execution manager

For the *end_of_reconfiguration* and the *reused_task* events (Line 5), the system checks the dependencies of the task that has been loaded or reused in order to identify if it can start its execution. To this end, the scheduler checks if all its predecessors have already finished their execution (Line 6). In that case, the task is ready to be executed (Line 7) and the scheduler starts its execution (Line 8). After that, it attempts to schedule a new reconfiguration (Line 10).

Finally, for the *end_of_execution* event (Line 11), the scheduler checks again if the reconfiguration circuitry is idle. If so, it looks for a new reconfiguration (Lines 12-14). Then, it updates the task-graph dependencies, decreasing the number of predecessors of each successor of the finished task (Line 15). Finally, the system checks if any of the tasks that are currently loaded in any RU are ready for their

execution by checking their dependencies (Line 17). In that case their execution takes place (Lines 18-20).

In Figure 4.9 there are three calls to the function *look_for_reconfiguration (&rec_sequence)* (Lines 3, 10 and 13). In this function, the scheduler must decide whether to schedule next reconfiguration in the reconfiguration sequence or not. And in that case, it must also decide the replacement victim. To this end, it takes into account the mobility of the task and the victim selected by the replacement policy.

```
void look_for_reconfiguration (&rec_sequence){
 1. new_task := next_task (rec_sequence);
 2. victim := apply_replacement ();
 3. IF (victim ≠ Ø){
 4.   IF (critical_candidate (victim) == TRUE AND
                          new_task.mobility > skipped_events){
 5.    skipped_events++;      /* initially 0 */
 6.   }ELSE{
 7.    load (new_task, victim);
 8.    delete_task (new_task, &rec_sequence);
 9.   }
10. }
}
```

Figure 4.10. Pseudo-code of the function *look_for_reconfiguration (&rec_sequence)*

Figure 4.10 presents the pseudo-code of this function. Basically, it attempts to schedule the reconfiguration of the first task in the reconfiguration sequence (Line 1). Initially the scheduler selects a victim applying a replacement policy that we have developed (Line 2, this policy will be explained in the following subsection). If all the reconfigurable units are busy, the replacement is not possible (Line 3). Otherwise (Lines 4-9), the function checks if the victim is critical and if the mobility of the task to load is greater than the number of total skipped events at that moment (Line 4). In that case, the system does not carry out the reconfiguration but it only increases the number of skipped events (Line 5). Otherwise, the function triggers the reconfiguration of *new_task* replacing the

selected victim (Line 7) and removing *new_task* from the reconfiguration sequence (Line 8).

Figure 4.11 describes in detail an example of the management of the execution of a task graph with five tasks in a system with three reconfigurable units. For the sake of simplicity, the weights of the tasks and the sequence of reconfigurations are already calculated. In addition, the mobility of all the tasks is always 0, hence their reconfiguration takes place as soon as possible.

Initially the manager receives the information about the task graph that has to be executed and it generates a *new_task_graph* event. Since the sequence of reconfigurations is: 1-3-2-4-2, the manager starts the reconfiguration of Task 1, which is the first one in the sequence. Thus, the **replacement module** is invoked in order to decide the destination RU for Task 1; and it decides to load it in RU1. In fact, every time a task must be reconfigured in the system, this replacement module is invoked. However, this example does not give further details about how the replacement decisions are taken. Instead, next subsection "*The replacement module*" will provide a detailed description about how it works.

When Task 1 finishes its reconfiguration, RU1 generates an *end_of_reconfiguration* event. Then, the manager checks if Task 1 can start its execution. In this case it has no unresolved dependencies, hence its execution starts just after its reconfiguration finishes. In addition, the control unit starts the following reconfiguration (Task 3) since the reconfiguration circuitry is idle and there is at least one available RU. When the reconfiguration of Task 3 finishes, a new *end_of_reconfiguration* event is handled, and the manager starts the reconfiguration of Task 2. In addition it also checks if Task 3 can start its execution, but it is not possible since the execution of Task 1 has not finished yet. And as soon as the reconfiguration of Task 2 finishes, a new *end_of_reconfiguration* event is processed, but this time no new reconfiguration starts since there is no available reconfigurable unit. In addition, the execution of Tasks 2 and 3 cannot start, since there is still an unresolved dependency: Task 1 has not finished its execution yet.

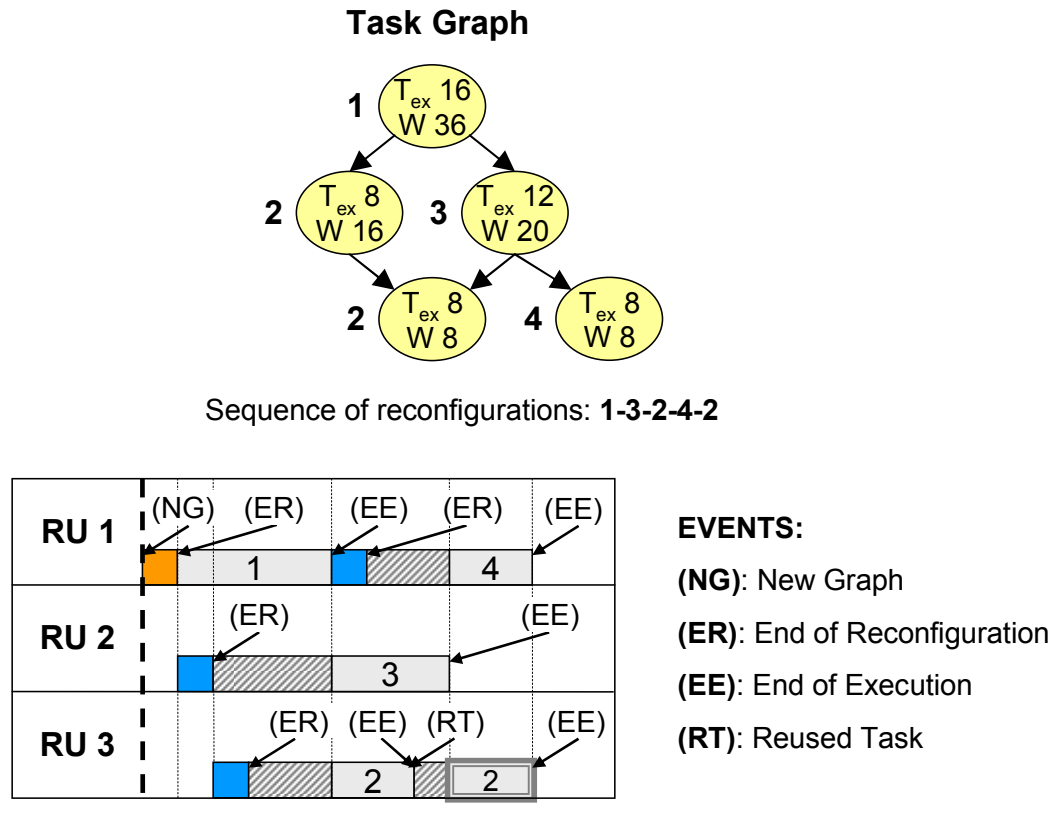

Figure 4.11. Example of execution of a task graph on our execution manager. The reconfiguration latency is 4 milliseconds

When Task 1 finishes its execution a new *end_of_execution* event is generated. The first step to handle is to update the dependencies, and then to try to start a new reconfiguration. In this case the manager starts to load Task 4 in RU1, since it is the only one that is available at that moment. After that, the manager looks for tasks ready to be executed, and identifies that Tasks 2 and 3 are loaded and have no unresolved dependencies. Hence, their execution is triggered.

When the reconfiguration of Task 4 finishes, the manager checks if it can start its execution, but in this case it is not possible because it has an unresolved dependency: Task 3 has not finished its execution yet. In addition it tries to schedule the reconfiguration of Task 2, but, again, it is not possible because all the reconfigurable units are busy. The next event is the end of the execution of Task 2. Again, the task-graph dependencies are updated and the manager tries to start a new reconfiguration. In this case it is possible to start the reconfiguration of the second instance of Task 2. However, since this task is already loaded in RU3, the system does not carry out this reconfiguration, but only generates a *reused_task* event, which is similar to an *end_of_reconfiguration* event. Finally, when Task 3 finishes its execution, the system updates the task-graph dependencies, and identifies that Tasks 2 and 4 can start their execution. Hence, both executions start.

The *end_of_execution* events generated after the completion of Tasks 2 and 4 does not trigger any further action, since there are no more tasks either to reconfigure or execute. Hence, after the execution of Task 4, the execution of the task graph completely finishes.

This example illustrates the benefits of the prefetch and replacement techniques. As Figure 4.11 shows, only one out of the five reconfigurations has generated a delay in the execution since one of the configurations has been reused (the second instance of Task 2) and the remaining three ones have been prefetched.

4.2.2. The replacement module

The example of the previous section did not provide any details about how the replacement module makes the replacement decisions during the run-time scheduling process; for instance, why Task 3 and 2 are loaded in reconfigurable

units 2 and 3, respectively. This section provides a detailed description about how the scheduler makes these decisions.

Basically, this module carries out the replacements according to a strategy especially designed for this system. We have called it *Look Forward + Critical* (LF+C) because it takes into account the *criticality* of the tasks and whether *they are going to be loaded in the near future* or not. We assume that the "near future" means "in the context of the graph currently in execution" because this is the only interval of time that the scheduler can analyze. (It is important to remember that the scheduler has no information about the task graphs that are going to come after the task graph currently in execution). We have designed a new replacement policy for three reasons:

- Firstly, conventional replacement policies, such as LRU, can easily fall into a configuration thrashing problem when the number of active tasks is greater than the number of reconfigurable units.
- Secondly, since the scheduler deals with task graphs, it has some information about the tasks that are going to be executed in the near future (within the involved task graph), and it can take advantage of that.
- Finally, the objective of conventional replacement policies is to improve the reuse. This is a good objective, but in our replacement policy it is only a secondary goal. In this case, the main objective is to improve the performance by attempting to reuse as many critical tasks as possible. In a nutshell, the goal of LF+C is not to reuse as much as possible, but to reuse as good as possible.

According to the information that the run-time scheduler receives about the tasks, it classifies the reconfigurable units into 6 categories, which are enumerated next:

- **Busy:** Reconfigurable units with a task in execution, or with a configuration that has recently been prefetched and that is waiting for execution. These ones are never selected for replacement.
- **Free candidates (FCs):** Reconfigurable units that do not have any loaded task. When the system is initialized, all the reconfigurable units are free.
- **Non-critical candidates (NCCs):** Reconfigurable units with non-critical tasks that are not going to be executed in the near future.
- **Non-critical reusable candidates (NCRCs):** Reconfigurable units with non-critical tasks that are going to be executed in the near future.
- **Critical candidates (CCs):** Reconfigurable units with critical tasks that are not going to be executed in the near future.
- **Reusable critical candidates (RCCs):** Reconfigurable units with critical tasks that are going to be executed in the near future.

The replacement technique assigns the maximum priority to the RCCs and the minimum priority to the FCs (the "*Busy*" candidates are a special category, since they cannot be replaced). Hence, LF+C attempts to replace those tasks that are not critical and/or are not going to be executed soon. For this purpose, it firstly considers the criticality of the tasks and then, their reusability. Thus, first of all LF+C always tries to replace a non-critical candidate rather than a critical one. If the possible replacement victims are either all critical or all non-critical, LF+C gives more priority those candidates that are going to be reused soon (*reusable* ones). If all of them are critical and belong to the same category (CC or RCC), the scheduler use the criticality value to select the victim. And if all of them had the same criticality (which also happens for the categories for FCs, NCCs and NCRCs), LF+C selects the first victim of the given type it finds.

Figure 4.12 depicts an example that illustrates the benefits of our replacement policy. The figure presents a system that executes three task graphs twice using

our LF+C replacement policy, and compares its results with two other well-known replacement strategies, **LRU** and **LFD**:

- **LRU (Least Recently Used)** replaces the reconfigurable unit that contains the task that was the least recently used for the last time with respect to the remaining ones.
- **LFD (Longest Forward Distance)** replaces the reconfigurable unit that contains the task that will be requested farthest into the future. This implies that, in order to use it, the system needs to know all the sequence of task graphs that will be executed in the system. Hence, it cannot be applied in our dynamic environment, in which we assume that the task graphs come for their execution in an unpredictable way. However, it can be used as a reference if we run a particular experiment where the sequence of task graphs to be executed is known in advance. This policy was originally proposed by Belady [Bela66] and guarantees the greatest reuse rate. In addition, we consider interesting the comparison between LF+C and LFD since in reconfigurable systems a high reuse rate involves eliminating a high amount of reconfigurable overheads.

Firstly, the Figure 4.12 shows that the well-known LRU policy does not reuse any task, since there are 7 tasks competing for 5 reconfigurable units. This is an example of configuration thrashing: the tasks are replaced shortly before they are used again, causing frequent misses. Hence, it provides the worst execution time: 92 milliseconds. Secondly, the LFD replacement policy (which is the optimal one regarding reuse) improves these results, by reusing 5 tasks and reducing the execution time to 84 milliseconds. Finally, the figure also shows what happens when the system uses LF+C, which reduces the total execution time to 80 milliseconds by reusing only 3 tasks.

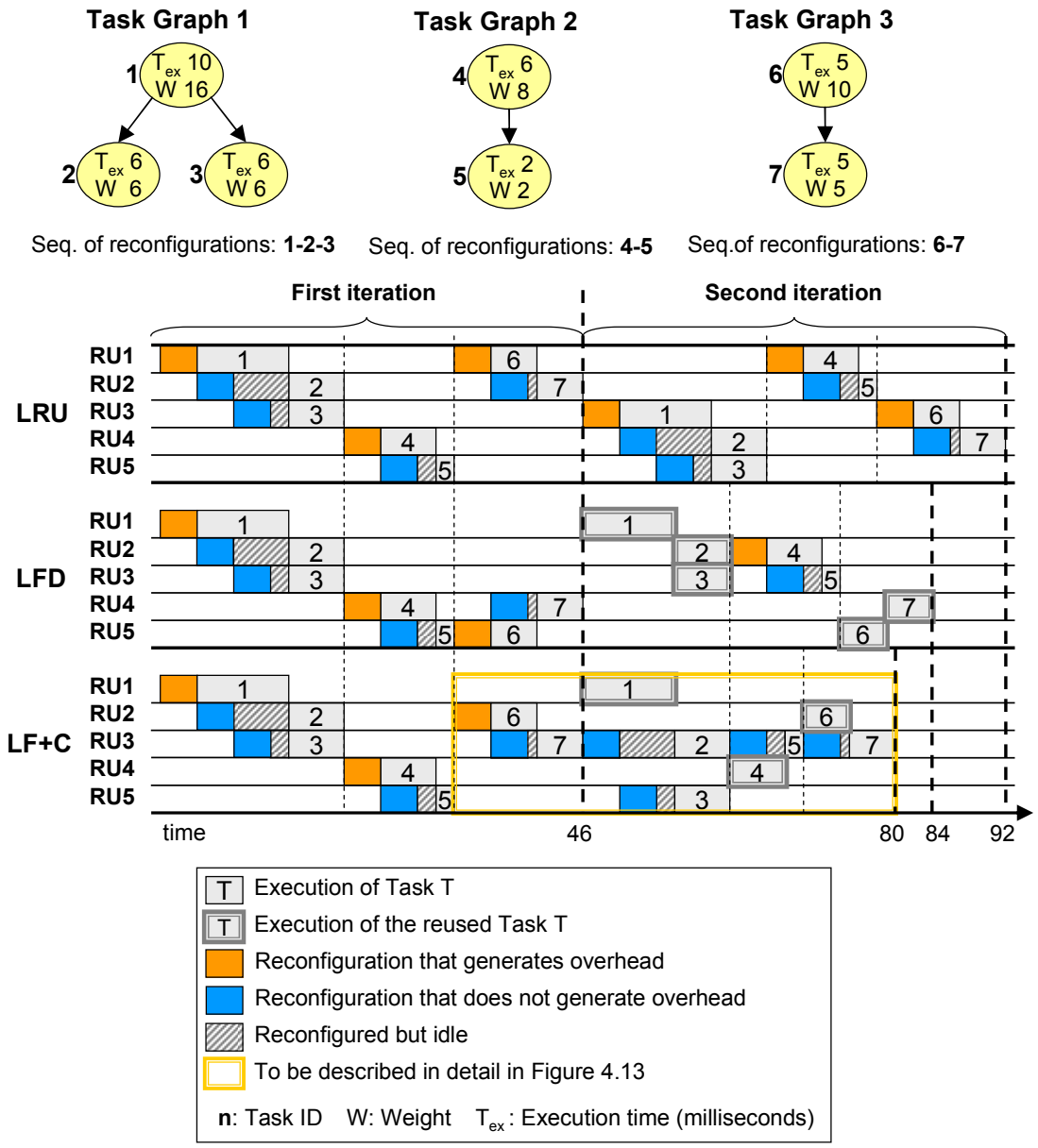

Figure 4.12. Execution of three graphs in a system with 5 reconfigurable units and 4 milliseconds of reconfiguration latency

As Figure 4.12 shows, for the first two task graphs the LF+C replacement policy simply selects a Free Candidate (FC) whenever a task must be reconfigured. The following replacement decisions (which are highlighted with a yellow rectangle) are described in greater detail in Figure 4.13.

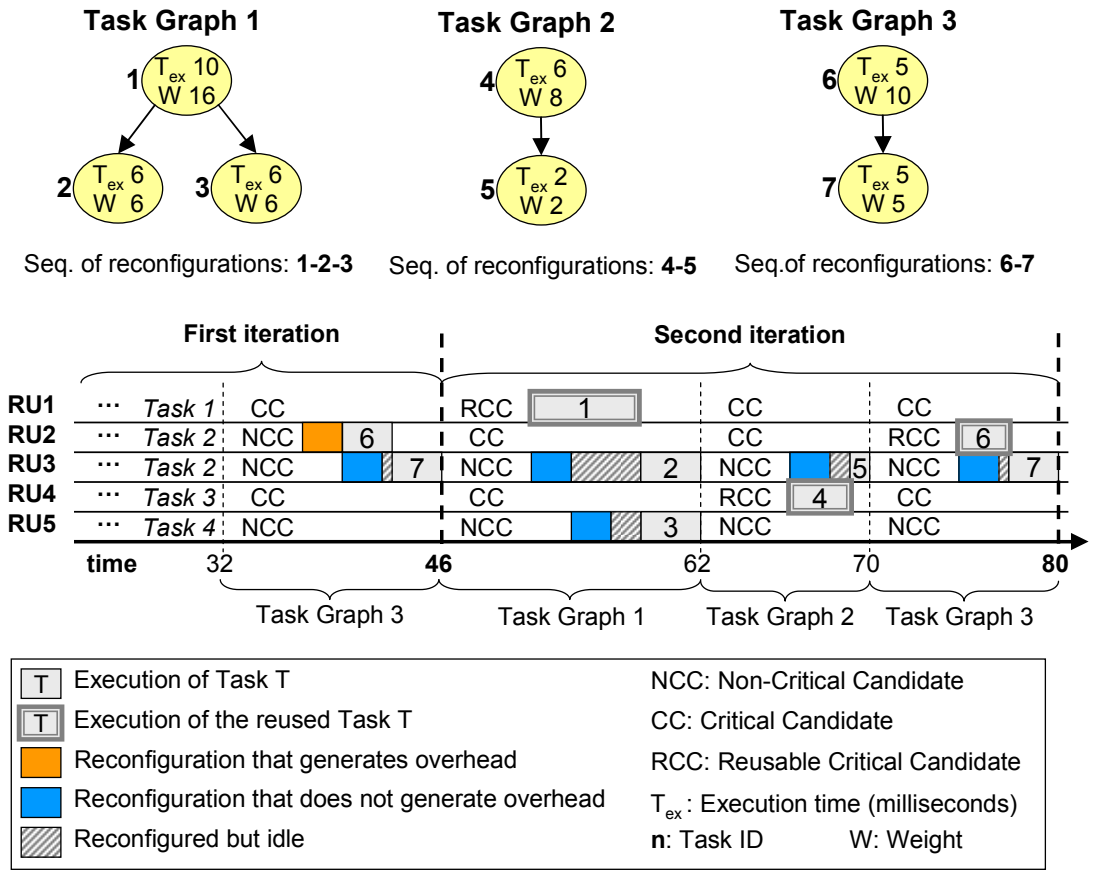

Figure 4.13. Replacement decisions made by the LF+C policy. Detail from Figure 4.12

Thus, when Task Graph 3 starts, RU1 and RU4 are critical candidates; whereas RU2, RU3 and RU5 are non-critical ones (none of them is reusable). Hence, in order to load the tasks of Task Graph 3, the replacement policy selects two non-critical candidates (RU2 and RU3), since our priority scheme gives more priority to CCs. Next, when Task Graph 1 is executed again, Task 1 is reused in RU1, since it is a reusable critical candidate (RCC). Then, the scheduler loads Tasks 2 and 3 in RU3 and RU5 respectively, since they are the only non-critical candidates that are available. In this way, the critical candidates (CCs) are not replaced. Note that during this second execution of Task Graph 1, only one task has been reused, but the other two reconfigurations do not generate any reconfiguration overhead, since they are not critical tasks. Immediately after the

second execution of Task Graph 1, Task Graph 2 is executed again. Its first node (Task 4), which is critical, is reused since it was not replaced during the execution of the two previous task graphs. Hence, its reconfiguration does not generate any reconfiguration overhead. For Task 5, the replacement policy selects the first NCC (in this case RU3) as victim. Finally, when Task Graph 3 is executed again, the scheduler reuses Task 6 and loads Task 7 in RU3, again a NCC.

As can be seen in the figure, LF+C does not achieve the optimal result regarding reuse. Whereas LFD reuses 5 tasks, LF+C only reuses 3. However, as it was explained before, reuse is only a secondary objective for our heuristic. The main objective is performance, and in this example LF+C achieves the optimal performance, hiding the latency of all the reconfigurations in the second execution of the task graphs. Using LF+C, the 6 graphs are executed in 80 milliseconds, 4 milliseconds faster than the execution time when using LFD and 12 milliseconds faster than a system that applies LRU. Hence, this example shows that the objective of LF+C is not only to reuse more, but also to reuse better.

As described above, the replacement policy tries not to replace the critical tasks, since in this way the scheduler increases the probability of reusing them in the future, thereby eliminating their reconfiguration overhead. The example of Figures 4.12 and 4.13 shows the convenience of following this scheme. In addition, the replacement policy gives more priority to those candidates that are going to be executed again in the near future (“reusable”), among either critical or non-critical candidates. This is the reason RCCs and NCRCs have more priority than CCs and NCs, respectively.

This also means that the scheduler gives more priority to CCs than to NCRCs, since the criticality of the candidates is taken into account before their reusability. Hence, the scheduler may choose to perform more reconfigurations of non-critical tasks rather than fewer reconfigurations of critical ones. This also has a good impact on the performance of the system. Figure 4.14 illustrates this idea by means of another example.

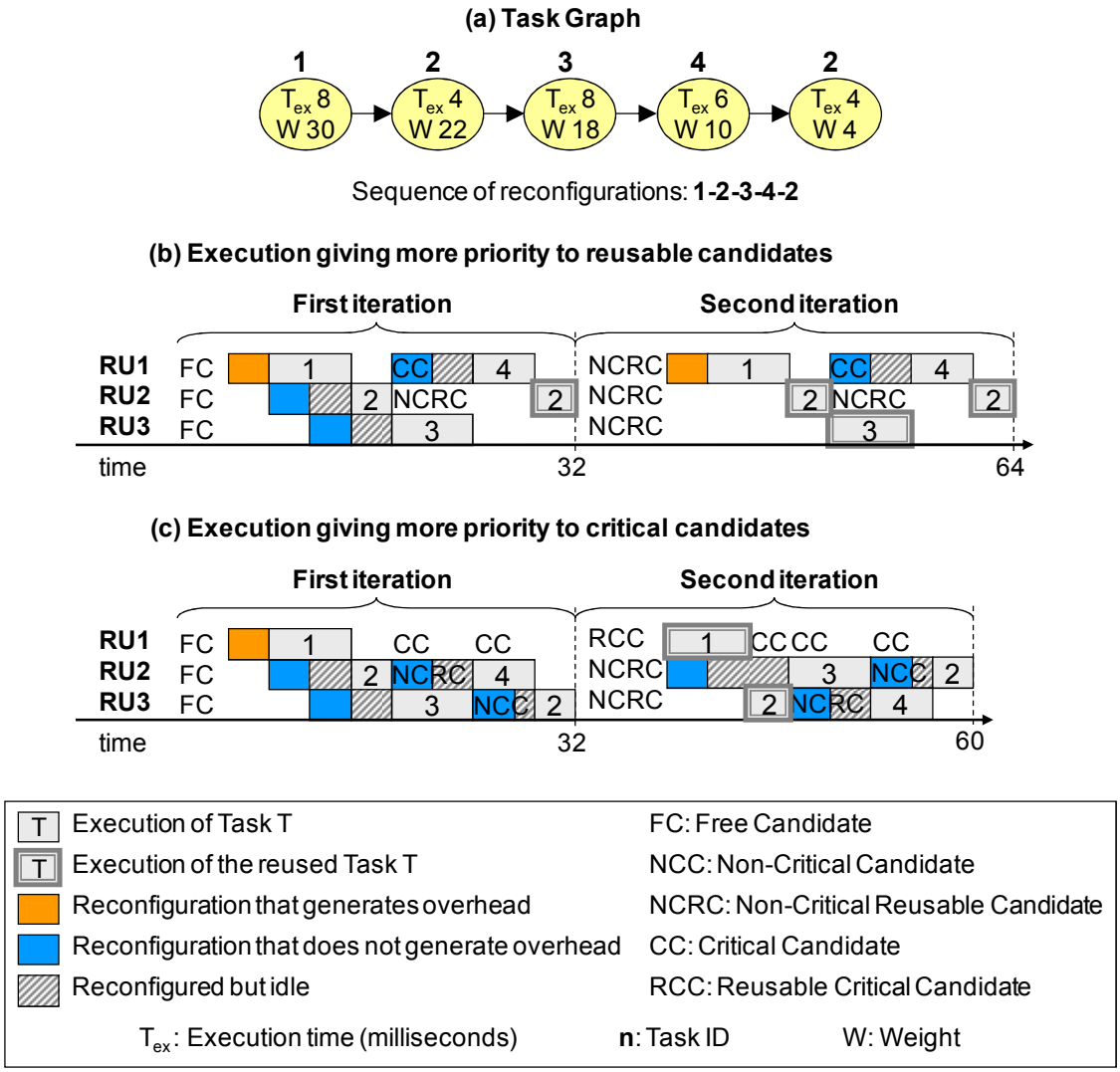

Figure 4.14. Motivational example of the convenience of giving more priority to CCs rather than to NCRCs. The reconfiguration latency is 4 milliseconds

Figure 4.14 shows the execution of a task graph in a system with three reconfigurable units. This task graph is executed twice. The figure compares what happens if the replacement technique assigns more priority to reusable candidates, and compares it with giving more priority to critical ones. It is also a good example to show how the scheduler uses the mobility of the tasks to delay some reconfigurations.

On the one hand, Figure 4.14.b) shows the results of giving more priority to reusable candidates. First of all, Tasks 1, 2 and 3 are loaded as soon as possible in RUs 1, 2 and 3, respectively, since the three RUs are free candidates and the mobility of these tasks is 0. The reconfiguration of Task 1 is the only one that generates any overhead, since it is the first one to be loaded. Then, after the execution of Task 1, the scheduler decides to delay the reconfiguration of Task 4 since the only available candidate at that time (RU1) is a CC, the mobility of Task 4 is 1 and it has not been delayed yet. Hence, the scheduler waits until the execution of Task 2 to load it on RU1 (which is a CC) rather than on RU2 (which is a NCRC), since in this case the scheduler decides not to replace reusable candidates. This makes possible to reuse Task 2 the second time it is executed, immediately after the execution of Task 4. Thus, the first execution time for the task graph of this example is 32 milliseconds.

However, when the task graph comes for its execution again, Task 1 has to be loaded again, since it was previously replaced by Task 4. This generates again a reconfiguration overhead of 4 milliseconds. Then, Tasks 2 and 3 are reused, since they were already loaded in the system. Similarly to its previous execution, Task 4 is loaded again in RU1 (replacing Task 1) and finally Task 2 is reused again. Thus, the second execution for the task graph takes again 32 milliseconds and this replacement policy does not hide the reconfiguration of Task 1, which is the only critical one, even though it reuses four tasks throughout the total execution: Task 2 (three times) and Task 3.

On the other hand, Figure 4.14.c) shows what happens if the replacement technique assigns more priority to critical candidates. This time, when the scheduler tries to load Task 4 for the first time, the replacement technique chooses a NCRC (RU2), rather than the critical one located in RU1. Note that this replacement is possible thanks that the scheduler decides to delay the reconfiguration of Task 4 again (similarly to Figure 4.14.b)). Finally, the second instance of Task 2 is loaded in RU3. In this way, when the task graph comes again, its first (and critical) task can be reused, and no reconfiguration overhead is generated this time. On the contrary, Task 3 and one of the two instances of

Task 2 have to be re-loaded in RU2, but this does not generate any additional latency since these tasks are not critical.

4.3. Conclusions

This chapter has described in detail the task-graph scheduler that has been developed in this doctoral thesis. Since the main objective pursued in this research work is to obtain efficient schedules in a highly dynamic environment, the proposed scheduler is a mixed design-time/run-time approach. The objective of the design-time phase is to obtain some useful information about the incoming task graphs that will be used at run time. More specifically, for each task of the task graph, three parameters are obtained: the **weight**, **criticality** and **mobility**.

The **weight** of a task represents how critical its execution is in the task graph. In other words, the tasks with greatest weights must be loaded earlier not to delay their execution and, in consequence, the execution of the whole application. Hence, once this information is obtained, the tasks are sorted decreasingly in order to determine the sequence of reconfigurations that will be carried out at run time. Thus, the tasks with greater weights are loaded in the system before.

The **criticality** of a task indicates whether its reconfiguration generates any temporal overhead or not. This information is used by the run-time scheduler to make the task replacements, which avoids deleting tasks that are critical or that are going to be reused soon.

Finally, the **mobility** of a task indicates to what extent the reconfiguration of that task can be delayed without generating any additional reconfiguration overhead. This information is used at run time in order to avoid falling into local sub-optimal schedules.

Once these three steps have been carried out, the run-time scheduling phase is carried out. Its objective is to steer the execution of the task graphs taking into account their internal dependencies, the information obtained at design time and the dynamic status of the system. For this purpose, in this phase the scheduler

exploits task prefetch and reuse in order to reduce the reconfiguration overhead during their execution.

This chapter has shown that the run-time phase comprises two modules that interact between them, namely **task-graph execution manager** and **replacement module**. The **execution manager** guarantees the correct execution of the task graphs taking into account their data dependencies and using the sequence of reconfigurations and the mobility of the tasks (this information was obtained at design time). For this purpose, it implements an event-triggered approach, in which the scheduling decisions are made in certain instants of time. On the other hand, the **replacement module** carries out the replacements according to a strategy especially designed for this system, which we have called *Look Forward + Critical* (LF+C). Basically, it uses the criticality of the tasks that was obtained at design time in order to avoid replacing the most critical ones or those that are going to be executed in the near future.

Since an important part of these computations are performed at run time, it is very important that the run-time scheduler does not generate too many temporal overheads in order to prevent performance degradations. This is the reason a hardware implementation of this phase has been developed and evaluated. The following chapter describes in detail the proposed hardware implementation and Chapter 6 evaluates its performance.

Your attitude, not your aptitude, will determine your altitude

Zig Ziglar

Chapter 5: Implementation details

The second important contribution of this doctoral thesis is the implementation of the run-time scheduler described in Chapter 4, using some of the reconfigurable resources existing in the target FPGA, a Xilinx™ Virtex-II Pro XC2VP30. In addition, an equivalent software version has also been developed, which consists in a software program that runs on one of the embedded Power PC 405 microprocessors of the target FPGA. The goal is to demonstrate that the proposed hybrid design-time/run-time approach is feasible in a practical scenario and how these two versions offer different trade-offs between the run-time scheduling overhead and the cost needed to implement the hardware circuit.

This chapter describes both implementations in detail. Thus, Section 5.1 gives implementation details about the hardware design proposed, whereas Section 5.2 does likewise with the equivalent software version. Both of them have been

developed using the Xilinx™ ISE and EDK 9.1.02i design tools. Finally, Section 5.3 summarizes this chapter with some final conclusions.

5.1. The proposed hardware design

This section describes in detail the proposed hardware implementation. Firstly, Subsection 5.1.1 gives low-level details about the run-time scheduler itself. Then, Subsection 5.1.2 describes how this hardware module has been integrated as a peripheral in a microprocessor-based embedded system that comprises a Power PC processor, the scheduler and a set of additional peripherals. This system has been used as a simulation platform to evaluate the performance of the scheduler, as well as to measure the run-time delays that the physical implementation generates.

5.1.1. Hardware scheduler: Low-level implementation details

Figure 5.1 shows a scheme of the developed hardware scheduler. It receives as inputs the sequence of reconfigurations and the information about the incoming tasks by means of the signals *Input Reconfigurations* and *Input Tasks*, respectively; as well as the external events, by means of the *new_task_graph* signal, which indicates that a *new_task_graph* event has been generated.

The scheduler stores the run-time events and the sequence of reconfigurations in the *Events Queue* and *Reconfigurations Queue*, respectively; and the task graph, in the *Task-Graph Table*. It also includes a set of *Reconfigurable Unit Module* for the RUs, each one of which contains two registers (for the current task and the state of the unit, respectively) and a small counter. The *Replacement Modules* makes the replacement decisions on these RUs and finally the *Control Unit* controls the overall operation of the scheduler by processing the run-time events (which are generated either by the RUs or externally) and makes the proper scheduling decisions. A detailed description of each one of these modules follows.

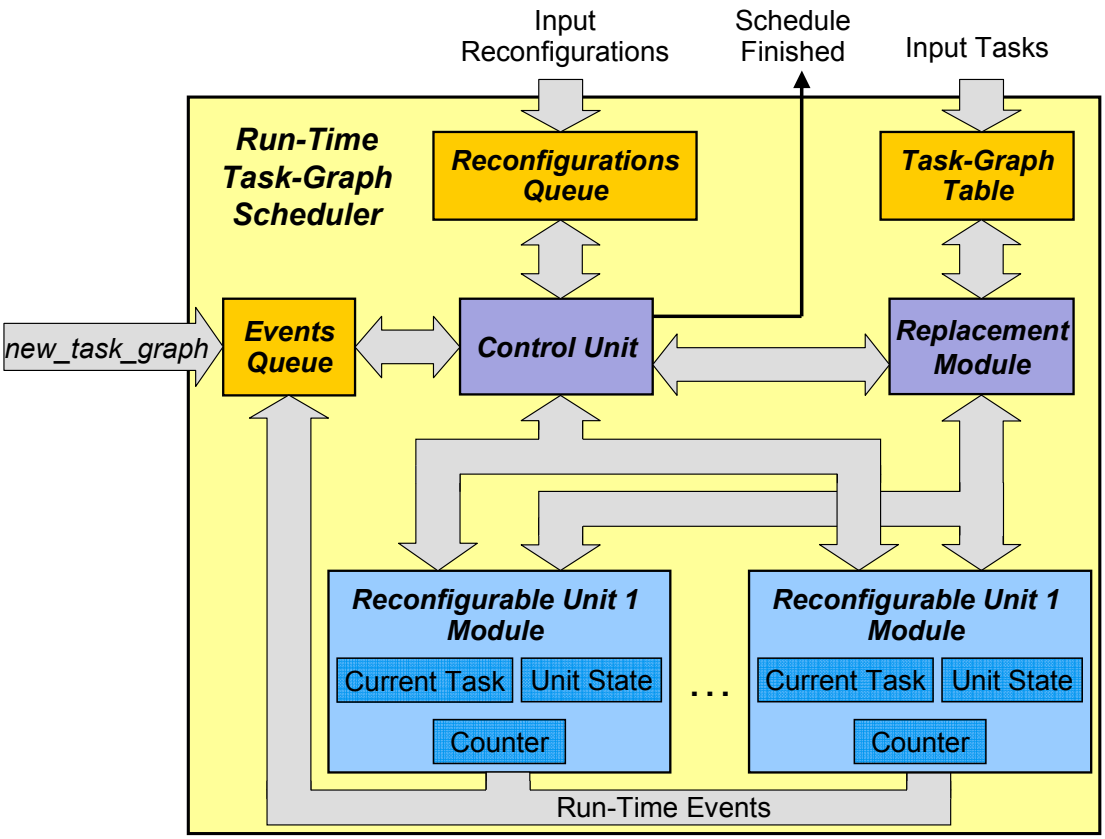

Figure 5.1. Scheme of the hardware implementation of the run-time scheduler

a) Reconfigurations queue

This module stores the sequence of reconfigurations that was obtained during the first step of the design-time phase of the scheduling flow. This sequence is stored in the queue when the information of a new task graph is received. Every time that the scheduler carries out a reconfiguration, it extracts the first task from this queue.

b) Modules for the reconfigurable units' information

As previously hinted in Chapter 2, we do not have an actual hardware multi-tasking system, but we just simulate the reconfiguration and the execution of

tasks on the RUs. Thus, instead of carrying out partial reconfigurations and executing the tasks, our hardware simulates the reconfiguration and execution times using programmable counters. *This is the only feature that is simulated in this hardware implementation.* Hence, instead of dealing with actual reconfigurable units, the scheduler includes a set of modules that characterize them. Thus, the reconfigurable unit *i* is identified in Figure 5.1 as *Reconfigurable Unit i Module*.

As Figure 5.2 shows, each one of these modules includes a counter, which is used to simulate the reconfiguration and the execution time delays, and two registers that store the information about the task currently loaded in that unit and the state of the unit, respectively. The *State Machine* manages the general operation of this module. The figure also depicts in greater detail the interaction between this module and the *Control Unit*.

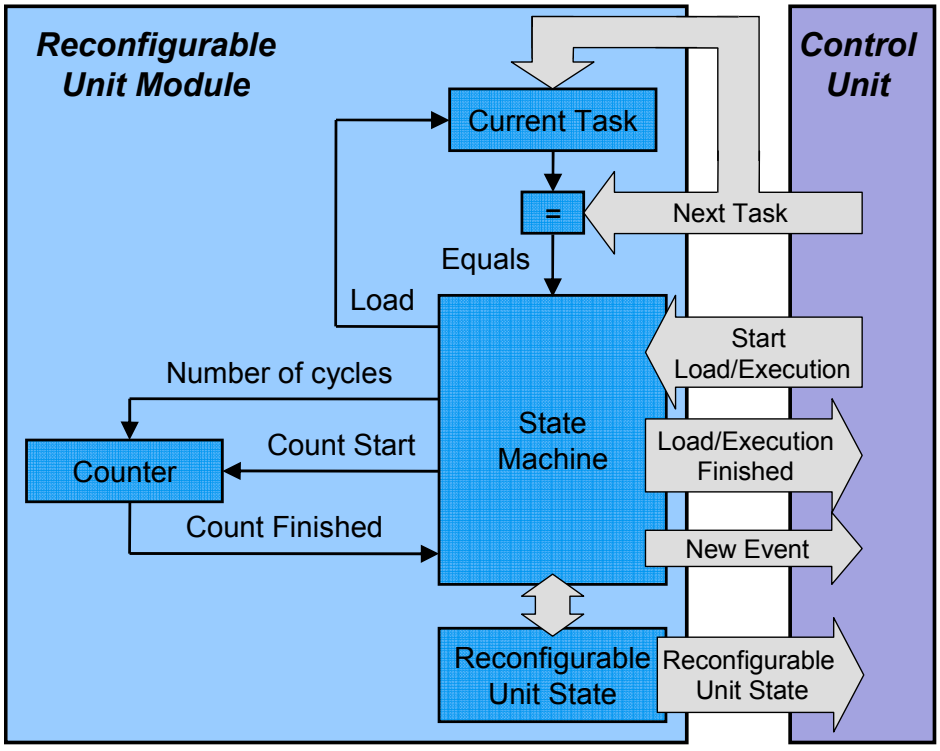

Figure 5.2. Scheme of the modules for the reconfigurable units' information

A task in a *Reconfigurable Unit Module* has a life cycle as shown in Figure 5.3. Its possible states are: *Idle*, *Reconfiguring*, *Ready to execute* and *Executing*. The *Control Unit* shown in Figure 5.2 manages the transitions depicted in 5.3 through the signals *Start Load/Execution* and *Load/Execution Finished*.

Basically, the *Reconfigurable Unit Module* complies with the orders that the *Control Unit* gives. Thus, when a new RU must be reconfigured, (transition *Idle – Reconfiguring*), the *Control Unit* sends a load order and the information about this task. Then, the *State Machine* loads this information in the register and simulates its reconfiguration latency by activating the counter. When this counter finishes, the transition *Reconfiguring – Ready to execute* takes place and the state machine acknowledges this through the signal *Load/Execution Finished*. The transitions *Ready to execute – Executing – Idle* are carried out in a similar way. Note that the *Control Unit* always knows the current state of the reconfigurable unit (by means of the *Reconfigurable Unit State* signal).

In addition, whenever the counter finishes a simulation of a reconfiguration or an execution time, the *State Machine* generates an *end_of_reconfiguration* or an *end_of_execution* event, respectively. An exception exists when the incoming task is already loaded in that RU, which is recognized by means of the equality comparator and the *equals* signal. If this happens, the *State Machine* skips the simulation of the reconfiguration time (*reconfiguring* state) and automatically switches to the *Ready to execute* state.

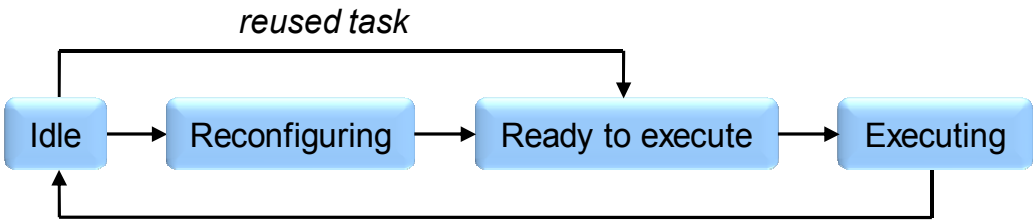

Figure 5.3. Life cycle of a task in our scheduler

c) Task-graph table

This table stores the information about the task graph to be scheduled and monitors the dependencies among the tasks. It supports four different operations: *Insertion*, *Check, Hit* and *Update*.

- ***Insertion***: This operation stores the information of a new task graph in the table. The latency of this operation is *NT* clock cycles, where *NT* is the number of tasks of the task graph.
- ***Check****:* This operation enquires about whether a task is ready to be executed or not, which happens when all its precedence constraints have already been met. The latency of this operation is just one clock cycle.
- ***Hit****:* This operation enquires about whether a task still exists in the table, which means that it has not been executed yet. This information is used by the *Replacement Module* in order to know whether a given task will be executed in the near future or not. The latency of the *Hit* operation is just one clock cycle.
- ***Update***: This operation updates the task-graph information when a task finishes its execution. First of all, the information about the task that has finished is deleted from the table; and afterwards the precedence constraints of all its successors are updated. To this end, all these successors are extracted and sequentially updated in order to decrease the number of remaining predecessors for each one of them. The latency of this operation in *2*NS+1* clock cycles, whereas *NS* is the number of successors of the task to be updated.

The information that is stored in each entry corresponds to a single task of the task graph. As Figure 5.4 shows, it is composed of: the number of predecessors

of the involved task, its number of successors and their IDs. This table has two customizable parameters: the size (in number of bits) of a task ID and the size of the binary number that determines the maximum amount of successors and predecessors that the task may have. Hence, depending on these parameters, the number of bits that each task entry uses changes.

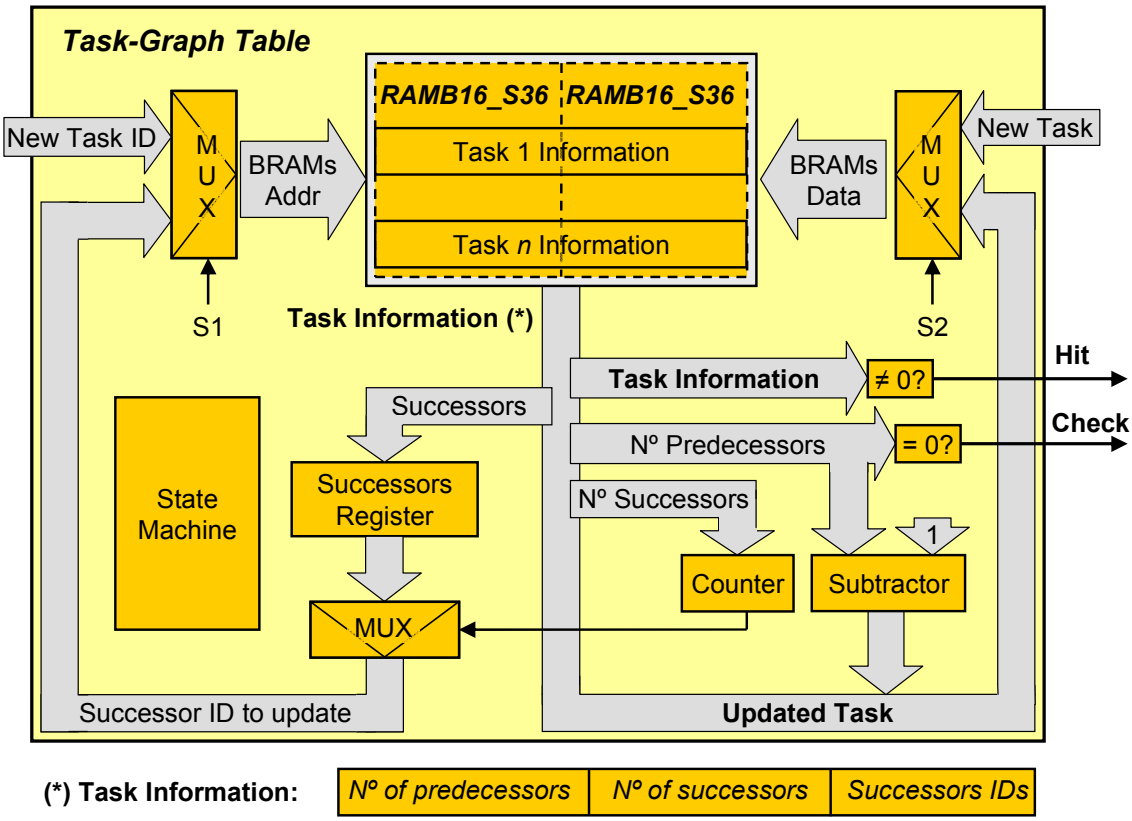

Figure 5.4. Table of task-graph dependencies

The information about the task graph is stored in two Block RAMs. More specifically, the type of these Block RAMs is *RAMB16_S36*, which is the name used in the VHDL code to instantiate a 16384-bit data memory with 512 entries that are 32-bit wide[11]. Both of them are used one next to the other in such a way that the information about a task graph is divided into both BRAMs and stored in

[11] *RAMB16_Sn*: http://www.xilinx.com/itp/xilinx5/data/docs/lib/lib0371_355.html

the same respective entries. Thus, the maximum theoretical width of a task entry is 64 bits. This width is enough space to store, for instance, tasks represented with 6-bits-wide task IDs[12] and 3-bits-wide fields to represent the numbers of successors and predecessors[13], since a single task line uses *3 (number of predecessors) + 3 (number of successors) + 2^3*6 (successors IDs) bits = 54 bits*. Hence, 64 bits is enough to represent all the tasks used in the experiments of this doctoral thesis, where there are 33 different tasks and all of them have at most four successors and predecessors, as it will be explained in the next chapter. In any case, the size of the task entry can be easily increased if needed adding as many additional BRAMs as necessary.

In addition, the information of Task *n* is stored in the n-th position of both BRAMs, which are accessed in parallel in order to retrieve that information. This additional constraint forces them to have as many entries as different tasks there may exist. However, there is plenty of space in the used BRAMs (512 entries) to store the 33 different tasks of our experiments.

As Figure 5.4 shows, the inequality comparator is used to implement the *Hit* operation, in such a way that if Task *n* exists in the table (the information stored in the corresponding entry is different from zero), the *Hit* signal is activated. For its part, the equality comparator implements the *Check* operation: if the current number of predecessors of the involved task is zero, then it is ready for its execution.

The rest of the hardware is used to implement the *Update* operation. The *State Machine* controls its general operation: First of all, the involved task is extracted from the BRAM; and its successors IDs and its number of successors are stored in the *Successors Register* and in the *Counter*, respectively. Then, the task is deleted from the BRAM. In order to update its dependencies, the counter iterates over all its successors by decreasing the number of predecessors of each one of them. For this purpose, these successors (which were stored in the *Successors Register*) are driven to the multiplexer, which sequentially selects the

[12] This means that, in this case, up to 2^6 tasks can be represented
[13] This means that, in this case, a task can have up to 2^3 successors

corresponding successor ID depending on the value of the *Counter*. This ID is then used to extract the information about the involved successor from the table, which takes one clock cycle. Once this information has been obtained, the subtractor updates it by decreasing the current number of predecessors by one (using the *Subtractor*). The updated information about the selected successor is stored again in the same entry of the task table (which takes again one clock cycle) and the counter is decremented for the next iteration. Hence, each iteration is carried out in two clock cycles. This is repeated for each successor of the finished task, hence the whole iterative process is carried out in 1 (the first deletion) + 2**Number_of_successors* (the iterative process) clock cycles.

Finally, note that the address and data inputs of the table are controlled by multiplexers, which decide whether what is written in the table is a new task or an updated existing one. The control signals of these multiplexers (*S1* and *S2*), as well as those of the register and the counter, are controlled by the *State Machine*.

d) Events queue

This queue is used to store the events that are generated throughout the scheduling process. Thus, whenever an event is generated, it is immediately stored here. However, it may happen that several events are generated at the same time (i.e. in the same clock cycle). This is a problematic situation, since this queue (and actually, all the queues of the system) have only one write port. For this reason, the *Events Queue* includes an *Arbiter* that guarantees that only one event is written in the queue at a time.

Figure 5.5 shows a scheme of how the *Arbiter* and the *Queue* (where the events are actually stored) interact. The input and output ports of the queue are controlled by the arbiter, which receives as additional inputs the event lines with the information that may be written there, as well as a set of *Write Requests* coming from the modules for the reconfigurable units. The output lines *Write*

Grants are the acknowledgements indicating that the corresponding events have been successfully written in the queue. Thus, if the arbiter decides to write the event coming from *Events[i]* (0 < i < number of reconfigurable units), it activates the lines *Load* and *Write Grants[i]*. Hence the selected event is driven to the *Queue* through the *Event Out* signal.

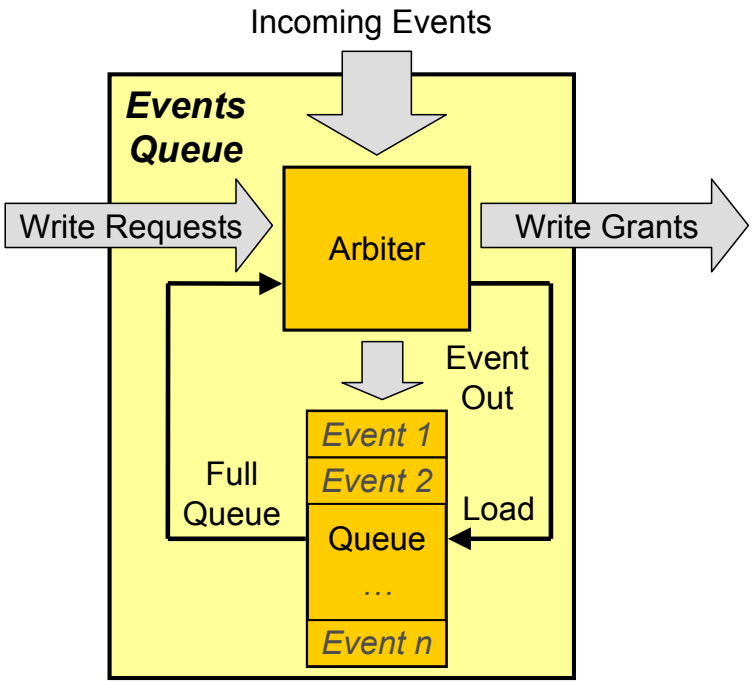

Figure 5.5. Scheme of the *Events Queue*

e) Replacement module

This module decides at run time in which reconfigurable unit to load the incoming tasks, taking into account their criticality and if they can be reused in any reconfigurable unit or not. It implements the *Look Forward + Critical* (LF+C) replacement technique developed in this doctoral thesis, which was described in detail in the previous chapter.

This module comprises a hardware that identifies the candidates (the *Candidates Identifier*) and a *Controller*, which iterates over the reconfigurable units and makes the replacement decisions. Figure 5.6 depicts the inner structure of the *Replacement Module*, focusing on the implementation details of the *Candidates Identifier*, which interacts with the *Task-Graph Table* and with the modules for the reconfigurable units. As shown in the figure, the *Replacement Module* interacts with the *Control Unit* in order to obtain the information about the available candidates whenever a task has to be loaded. A big part of this interaction is performed by the *Controller*, which will be described in detail below.

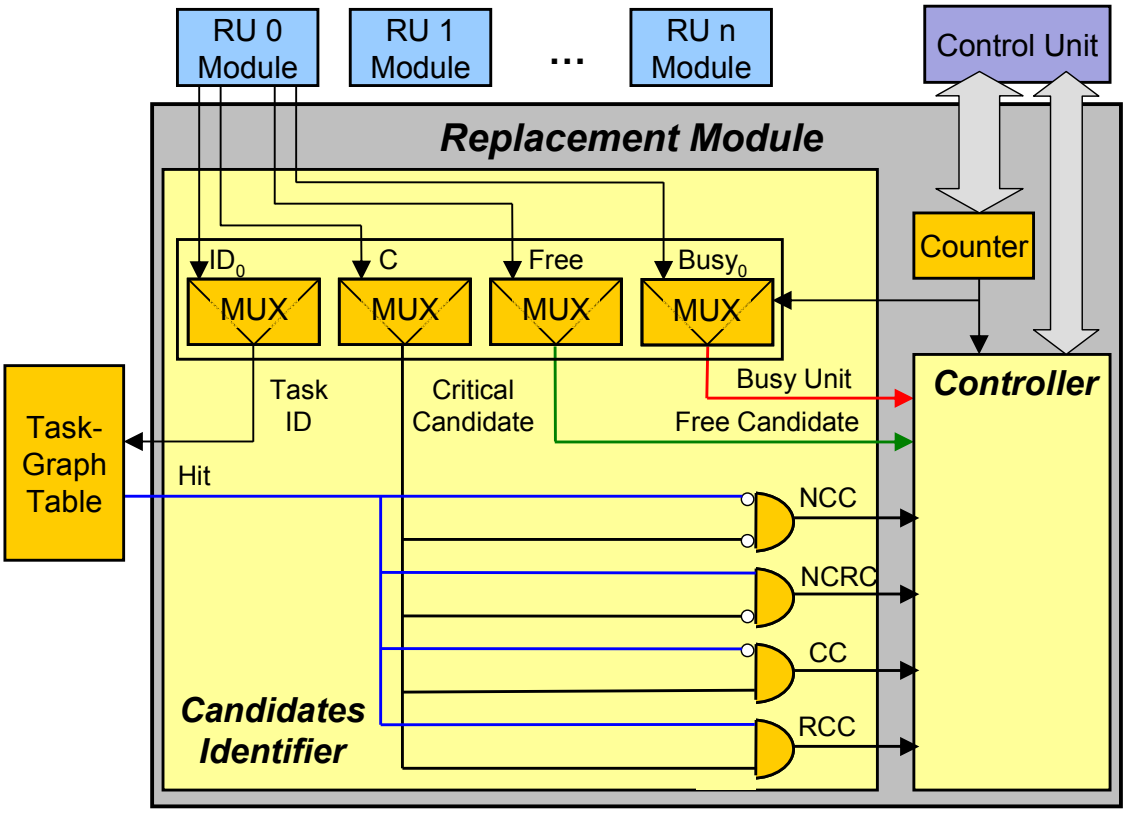

Figure 5.6. Scheme of the *Replacement Module* and detailed information of the hardware that identifies the candidates

As shown in the figure, the *Candidates Identifier* comprises four logic gates and four multiplexers, which are controlled by a *Counter*. Its value oscillates

between 0 and *num_reconfigurable_units* - 1 and in each clock cycle, it selects the information about the corresponding reconfigurable unit through the multiplexers. Then, the result of the *Hit* operation of the *Task-Graph Table* reports whether that task is going to be executed in the near future or not. Finally, the 'AND' gates identify the type of candidate that each reconfigurable unit is, according to the LF+C replacement policy:

- ***Busy***: If the RU is in the *Busy* state.
- ***Free***: If the *Free* signal of the RU is active.
- ***Non-Critical Candidate (NCC)***: If the candidate is not critical and the *Hit* operation returns a 0 (the candidate is not going to be used in the near future).
- ***Non-Critical Reusable Candidate (NCRC)***: If the candidate is not critical and the *Hit* operation returns a 1.
- ***Critical Candidate (CC)***: If the candidate is critical and the *Hit* operation returns a 0.
- ***Reusable Critical Candidate (RCC)***: If the candidate is critical and the *Hit* operation returns a 1.

On the other hand, Figure 5.7 depicts again the inner structure of the *Replacement Module*, but this time focusing on the *Controller*. The interaction between the *Replacement Module* and the *Control Unit* is now described in detail: Basically, the *Control Unit* makes a request to the *Replacement Module* to load a task and the latter responds with a reconfigurable unit identifier, following the LF+C replacement policy.

The counter that appears in Figure 5.7 is the same as in Figure 5.6. It iterates between 0 and *num_reconfigurable_units* - 1 (in the worst case) and depending on the information obtained by the *Candidates Identifier*, it properly updates the

registers *Reg_NCC*, *Reg_NCRC*, *Reg_CC* and *Reg_RCC*. These four registers store the position of the first candidates of each type found, respectively. A priority encoder (*ENC*) indicates whether the task to be loaded can be reused or not (signal *A* activated).

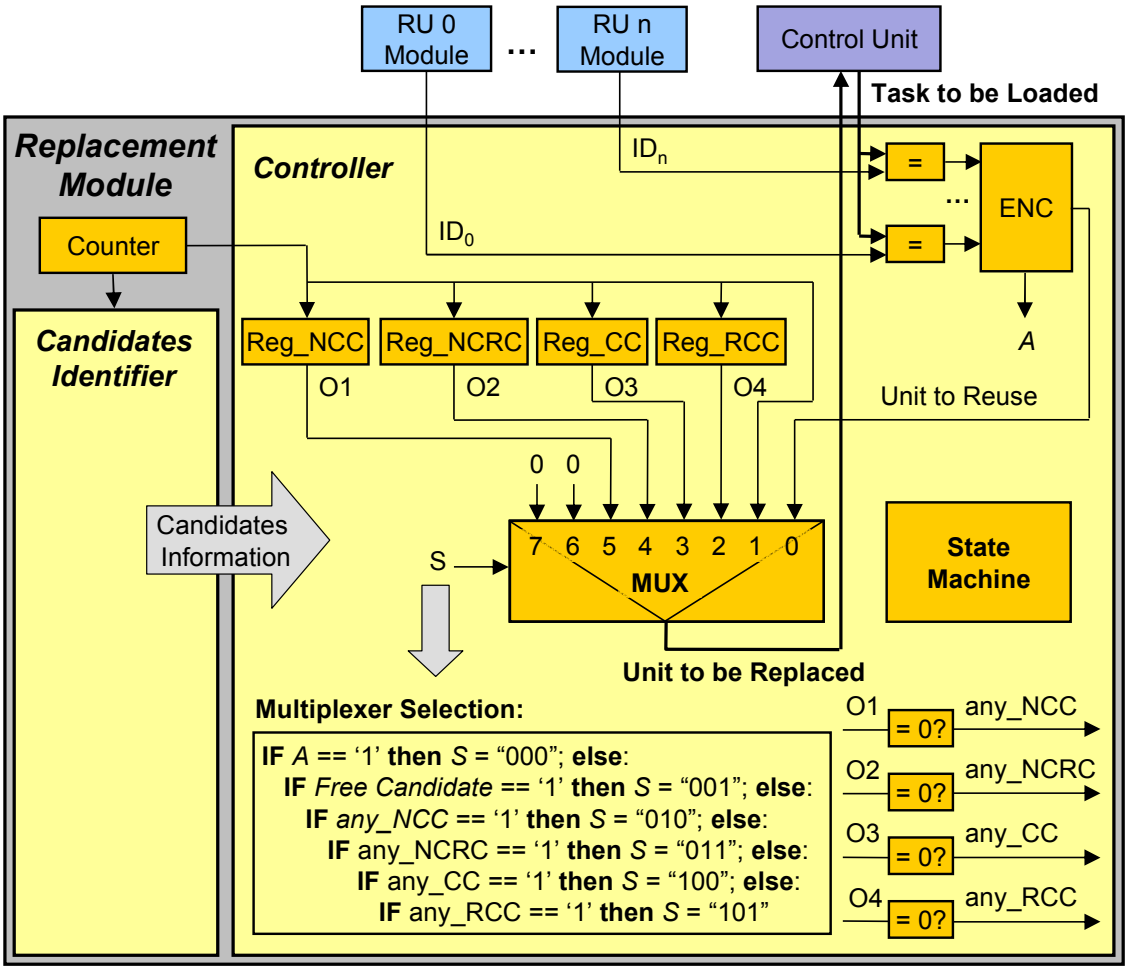

Figure 5.7. Scheme of the *Replacement Module* and detailed information of the *Controller* sub-module

There is also an 8-to-1 multiplexer that selects the candidate. It is connected to the outputs of the registers, but also to the outputs of the counter and the priority encoder. The output of the latter is directly connected to the first input of the multiplexer. It indicates the first reconfigurable unit found on which the task to

load can be reused. This reconfigurable unit is automatically selected in the case such a candidate exists. The selection of the first Free Candidate (FC) found (the output of the counter) works in a similar way, but without using an encoder. This is the reason there is no register to store these types of candidates, on the contrary to the remaining ones, because as soon as the controller identifies a FC, it selects it without analyzing the remaining reconfigurable units.

The selection of the multiplexer (*S* signal) follows the rules depicted in the figure. It implements the LF+C replacement policy, giving more priority to the reuses; and then, to the FCs, NCCs, NCRCs, CCs and RCCs, in this order (note that the signal *Free Candidate* is part of the *Candidates Information* signal that comes from the *Candidates Identifier*, see Figure 5.6). Finally, the output of the multiplexer indicates the reconfigurable unit to be replaced and is the output of this module.

The *Controller* also includes a *State Machine* that controls the hardware that has been just described. Basically, if the task to load cannot be reused, it iterates over all the reconfigurable units and stores the proper information on the registers, depending on the type of candidate that has been found. When all the reconfigurable units have been analysed, the output of the multiplexer identifies the replacement victim. However, if the controller finds a FC, it aborts the search process and that candidate is selected as the replacement victim.

f) Control unit

This module, which is depicted in Figure 5.8, is composed of two sub-modules: The *State Machine* extracts the run-time events from the *Events Queue* and triggers the proper management actions that were described in the previous chapter. Next, the *Hardware for the Skip-Events Feature* takes into account the mobility of the task to be loaded and the available replacement candidates in

order to decide whether to delay that reconfiguration or not. A detailed description of the *Hardware for the Skip-Event Feature* follows.

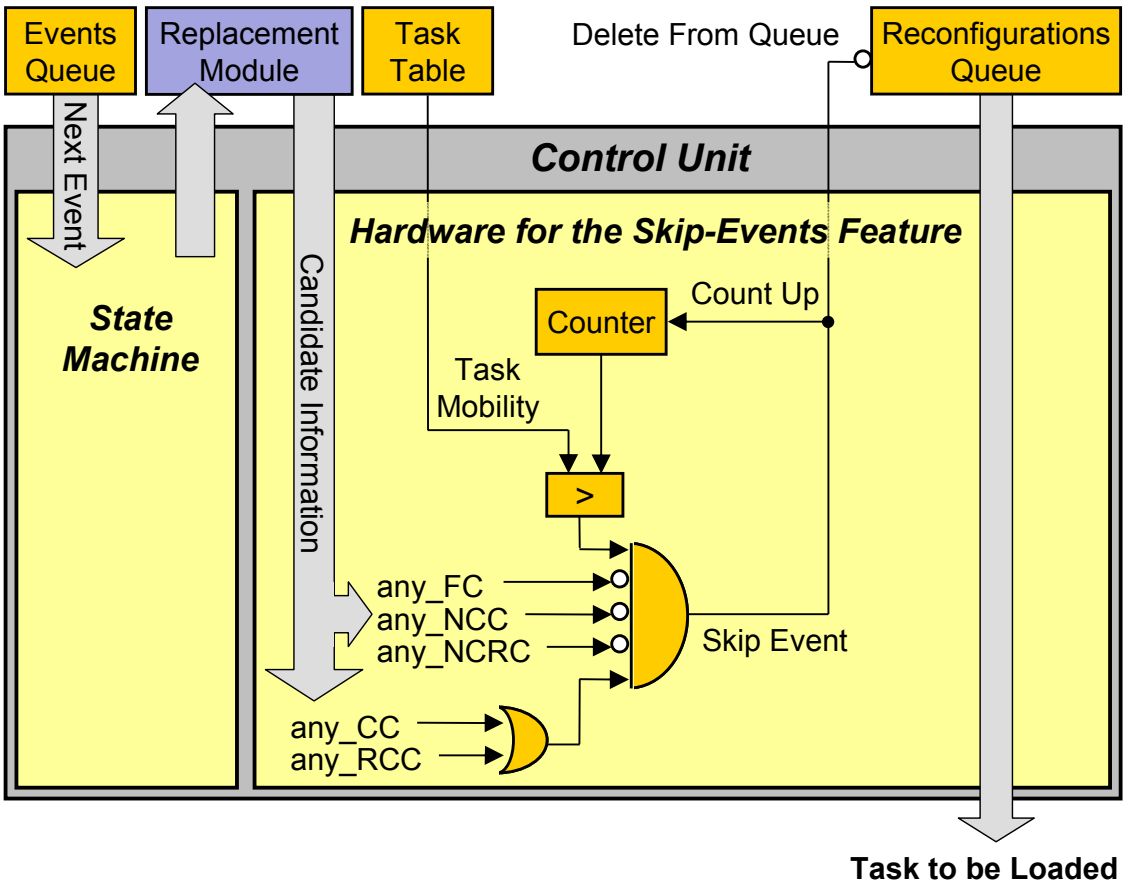

Figure 5.8. Scheme of the *Control Unit*, focusing on the hardware support to implement the *Skip-Events* feature

Basically, this sub-module obtains the first task stored in the *Reconfigurations Queue* and retrieves the information about the replacement candidate that the *Replacement Module* has selected as victim for this task. Next, depending on this information, this sub-module decides whether to delay the involved reconfiguration or not. For this purpose, there is a counter that stores the number of events skipped so far and a comparator that identifies if the involved reconfiguration can be delayed without introducing overheads (i.e. if the mobility assigned to the task is greater than the number of events skipped so far). If this

happens and all the replacement candidates are critical, the `AND´ gate decides to delay that reconfiguration. This means that the counter for the skipped events is incremented and the involved reconfiguration is not deleted yet from the *Reconfigurations Queue* (signal *Skip Event* active).

Finally, when the reconfiguration of that task starts, this task is removed from the *Reconfigurations Queue*.

5.1.2. Hardware scheduler: Simulation platform for performance evaluation purposes

The previous subsection described in detail the proposed hardware implementation of our scheduler. The next step in the implementation process was to integrate it in a hardware microprocessor-based embedded system, which includes a microprocessor and several peripherals. The goal of adding this layer of hardware is to create a testing platform able to evaluate the performance of the scheduler with clock-cycle accuracy.

In order to build this system, the Xilinx™ EDK 9.1.02i design tool has been used. It is a suite of tools and hardware blocks that can be used to design a complete embedded microprocessor-based system for implementation in a Xilinx™ FPGA. In addition to these hardware modules, this tool includes a set of drivers and libraries for the embedded software development, as well as a GNU compiler and debugger for C/C++ targeting the MicroBlaze and Power PC architectures. A detailed high-level description of the proposed embedded system follows.

a) High-level description of the simulation platform

Figure 5.9 depicts a scheme of the mentioned microprocessor-based system. It has been deployed in our Virtex-II Pro, which communicates with the external

PC via serial communication. As can be seen in the figure, the run-time scheduler is a hardware module integrated in the embedded system, which comprises a Power PC processor and a set of slave peripherals. Depending on the peripheral, they are attached either to a Processor Local Bus (PLB) or to an On-Chip Peripheral Bus (OPB).

The PLB[14] is a bus specifically designed to connect a processor to memories and other peripherals that need high data rates. It has some features that make it better suited than other buses (such as OPB) for high-speed accesses. Among its main features are 64-bit data transfers, simultaneous read and write and low-latency transfers. In addition, it is directly attached to the Power PC; hence there is no need for a bus bridge in between, which generates an additional delay. For its part, the OPB[15] is specifically designed to be as simple as possible. In other words, it is intended to use fewer logic resources and the communication interface is considerably simpler with respect to the PLB. This does not mean that the OPB is much slower than PLB, but the theoretical maximum throughput is considerably higher for the latter. Hence, the OPB continues being more suitable for peripherals that do not require high throughput. Xilinx™ EDK provides Intellectual Property (IP) cores for both buses and for the *PLB/OPB Bridge*; hence they have been used in the hardware design.

As shown in the figure, the system also includes two Block RAMs (BRAMs) that are used to store the processor instructions and data (*Data BRAM* and *Instruction BRAM*, respectively). Both of them are attached to the PLB by means of a BRAM controller, which is the communication interface between the memory and the bus.

As for the OPB, it is connected to the PLB by means of a bridge (*PLB/OPB Bridge*), since the Power PC does not include a communication interface to directly connect itself to the OPB. There are two peripherals attached to this bus:

[14] "*128-Bit Processor Local Bus Architecture Specifications Version 4.7*", IBM Corporation, 2007. Available at: https://www-01.ibm.com/chips/techlib/techlib.nsf/pages/main
[15] "*On-Chip Peripheral Bus, Architecture Specifications, Version 2.1*", IBM Corporation, 2001. Available at: http://ens.ewi.tudelft.nl/Education/courses/et4351/opb_ibm_spec.pdf

a universal asynchronous receiver-transmitter (the *RS232 UART*) and a programmable counter (the *OPB Timer*). The UART allows establishing a serial communication between the FPGA and the external PC. In this case, it implements the Recommended Standard 232 (RS232), which is a standard for serial binary single-ended data and control signals between the data transmitter and the receiver [EIA85]. Thus, the external PC is able to capture and display the results generated by the scheduler. The *OPB Timer* is used to experimentally measure the run-time delays generated by the hardware scheduler.

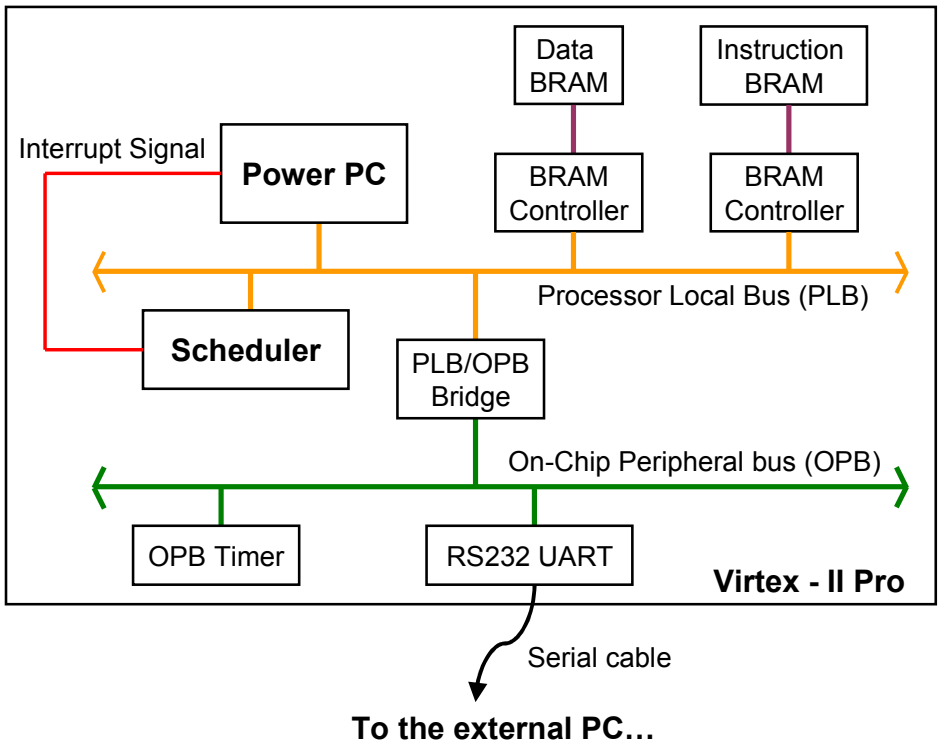

Figure 5.9. Microprocessor-based system where the hardware scheduler has been integrated

Our scheduler has been built as a peripheral that is attached to the PLB bus. It is a slave peripheral; hence its operation is entirely controlled by the processor. In addition, in order to inform the processor that a scheduling process has finished, the scheduler generates an interrupt as soon as it finishes its

computations. This is the reason there is an additional *Interrupt Signal* connecting the Power PC and the *Scheduler*. Finally, there is a piece of software code running on the Power PC that controls the proper operation of the system. It has been written in the C programming language and specifically compiled for the Power PC architecture. This code easily controls all the existing peripherals in the system, thanks to the drivers that the EDK tool includes.

The system works as follows: First of all, the processor sends the information about the task graphs (which is hard-coded for simplicity) to the scheduler. Then, it activates the *OPB Timer* to start the execution time measurement and sends a *new_task_graph* event to the scheduler, which initiates it. When the scheduler finishes, it informs the processor by activating the interrupt signal. And when this happens, the processor stops the timer and reads its value in order to know the elapsed time during the scheduling process. The difference between this time and the ideal execution time assuming no scheduling delays (which can be easily obtained with a software simulation), is the run-time penalty generated by our hardware implementation of the run-time scheduler. Hence, this scheme allows evaluating with clock-cycle precision the penalties generated due to the scheduling of the graphs considering both the computations performed and the communications among the scheduler and the processor.

Our peripheral has been built using a wizard that allows integrating the developed VHDL code as a part of an initially empty peripheral in order to customize it. The peripheral that the wizard creates by default implements the communication interface with the PLB bus and includes an initially empty module that can be filled up with the code developed by the user.

b) The developed peripheral

Figure 5.10 shows the inner structure of the developed peripheral. Basically, it is composed of a customized hardware block (the *User Logic*) that is connected to the *Intellectual Property Interface* (IPIF) by means of an *Intellectual Property*

Interconnection (IPIC)[16]. The IPIF is, in turn, interconnected with the PLB and implements the bus communication protocol. On the other hand, the *User Logic* block includes the source files with the VHDL code of the hardware scheduler.

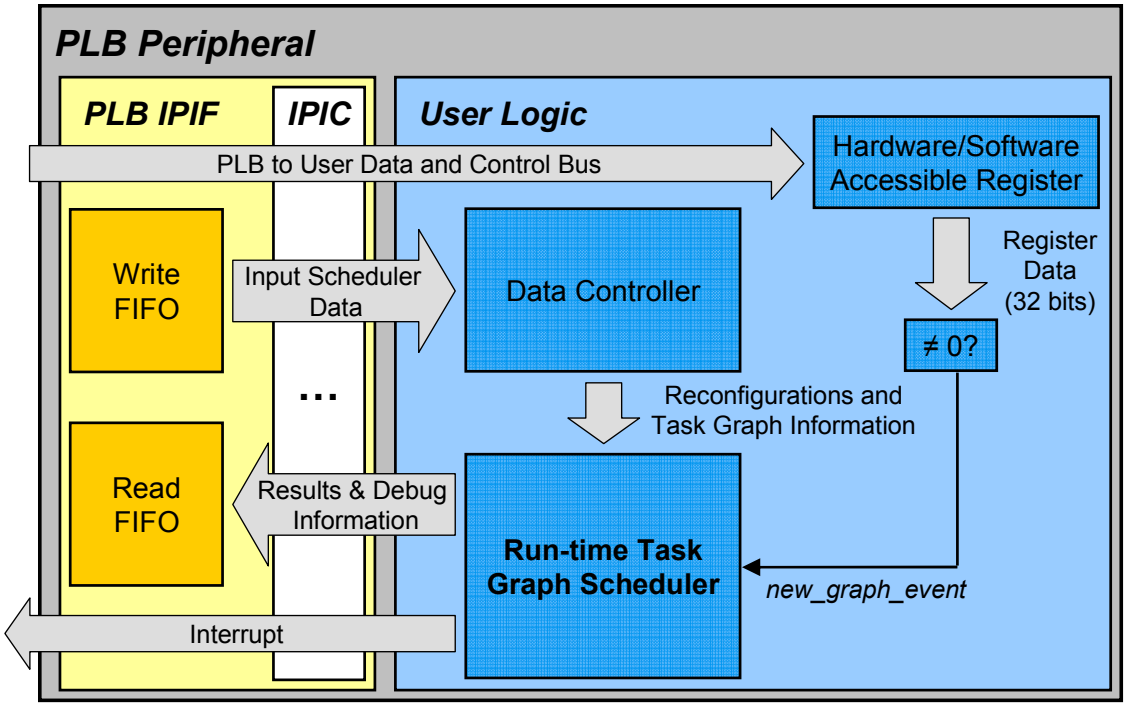

Figure 5.10. Scheme of the developed PLB peripheral in EDK and inner structure of the *User Logic*

The IPIC comprises the set of input/output signals of the *User Logic*. It includes a *PLB to User Data and Control Bus*, which is used to retrieve data directly coming from the bus, and which is connected to a register located inside the *User Logic* (the *Hardware/Software Accessible Register*). The IPIC also includes a bus for the input data of the scheduler (*Input Scheduler Data* signal) and for the results and the debug information (*Results & Debug Information* signal); as well as an interrupt line (*Interrupt* signal).

[16] *PLB IPIF (v2.02a)*:
http://www.xilinx.com/support/documentation/ipembedprocess_coreconnect_plb-ipif.htm

On the other hand, the IPIF contains a *Write FIFO*, which is used to store the information about the task graph and the sequence of the reconfigurations, and a *Read FIFO*, which is used to store the final results of the schedule and the debug information. Both the PLB IPIF and the IPIC contain many more modules and interconnection signals than those that are shown in Figure 5.10. These additional elements have been omitted in the figure for simplicity. However, a detailed description of the PLB IPIF module and the IPIC can be found in Appendix A, at the end of this PhD dissertation.

Figure 5.10 also shows the structure of the *User Logic* block. It includes the hardware implementation of the run-time scheduler (previously described), a register and a data controller.

The *Data Controller* is used to interpret the data stored in the *Write FIFO* and to initialize the *Reconfigurations Queue* and the *Table of Task Dependencies*, which are located in the scheduler. The processor writes the information to the *Write FIFO* in the following format: first of all, the sorted sequence of reconfigurations (one FIFO line per reconfiguration). Then, a string of zeroes and finally, the information about the task graph (one FIFO line per task). The sequence of zeroes is used by the controller to know when the sequence of reconfigurations finishes and when the information about the task graph starts.

In EDK, a *User Logic* module can include one or several *Hardware/Software Accesible Registers*, which are a set of addressable registers either from the user software or from the hardware of the peripheral. The number of available registers and their width is customizable. They can be created either manually or with the EDK wizard to create peripherals. In the latter case, the wizard creates a piece of VHDL code inside the source file for the *User Logic*. However, this code can be later modified by the user without needing to run the wizard again. As Figure 5.10 shows, in this case the user logic includes one software accessible register that is 32-bit wide and it is directly connected to the PLB through the *PLB to User Data and Control Bus*, in order to get the information coming from the PLB bus. It is used in such a way that, as soon as something different from zero

is written there, the scheduler must start the execution of the task graph. This is equivalent to generate a *new_task_graph* event and is controlled by the inequality comparator of the figure.

5.2. The equivalent software version

This section describes in detail the proposed software implementation of the scheduler. It consists in a software program originally written in the C programming language and compiled for the Power PC architecture. This code runs in one of the embedded Power PC 405 microprocessors existing in the target Virtex-II Pro FPGA.

Firstly, Subsection 5.2.1 describes the microprocessor-based embedded system where this program has been deployed. As the embedded system presented in the previous section, this one has also been used as a simulation platform to evaluate the performance of the scheduler, as well as to measure the run-time delays that the physical implementation generates. Then, Subsection 5.2.2 gives details about the software scheduler itself.

5.2.1. Software scheduler: Simulation platform for performance evaluation purposes

Figure 5.11 shows a scheme of the microprocessor-based system where the software program runs. As the system described previously for the hardware implementation, this system has also been deployed in our Virtex-II Pro, which communicates with the external PC via serial communication. The main difference with respect to the previous system is the inexistence of a dedicated peripheral for the scheduler (this time it runs directly on the Power PC), a set of programmable counters that simulate the behaviour of the reconfigurable units and the *OPB Interrupt Controller*.

As can be seen in the figure, there are as many programmable counters (*OPB Timers*) as reconfigurable units there are in the system. As in the hardware implementation, they simulate the reconfiguration and execution times of these

units. These counters interact with the processor by means of the interface that EDK provides to start/stop/resume them, but also by means of an interrupt generation and handling mechanism. This allows the processor to know whether the simulation of a specific reconfiguration or execution time has been performed successfully, and to carry out the proper scheduling actions. Each *OPB Timer* has two internal counters, which are used to simulate the reconfiguration and execution times of the tasks, respectively. Thanks to the communication interface that EDK provides, it is possible to work out which one of these two counters has generated the interrupt. Hence this allows identifying if an interrupt has been generated by an *end_of_execution* or by an *end_of_reconfiguration* event.

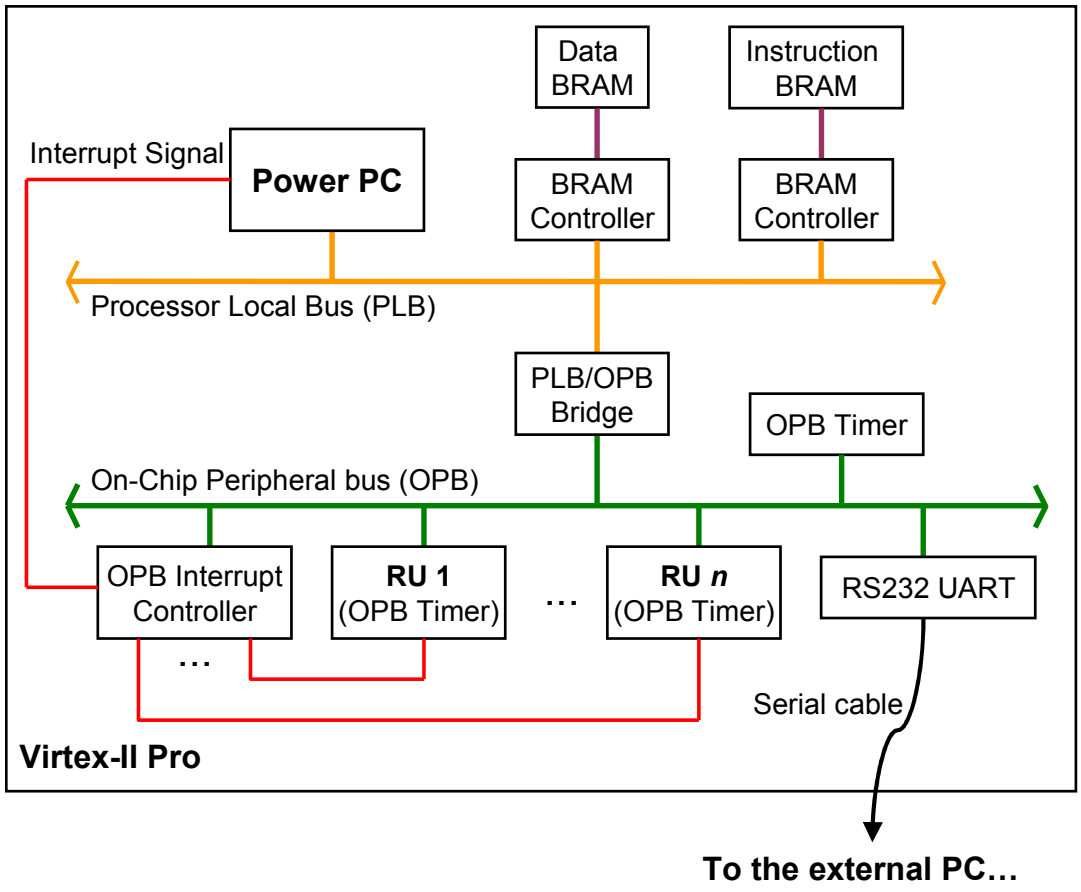

Figure 5.11. Microprocessor-based system used where the software scheduler has been integrated

The Power PC has two input interrupt signals: for the critical and non-critical interrupts, respectively (they do not appear in Figure 5.11 for simplicity). Hence, if there are more than two peripherals that can generate interrupts, it is necessary to add an additional module to be able to handle them all. This module is referred in the figure as an *OPB Interrupt Controller*, which concentrates up to 32 interrupt inputs from peripheral devices to a single output one, which is finally connected to the processor. In this case, the inputs for the interrupt controller are the interrupt signals coming from all the *OPB Timers*..

Finally, the rest of the modules existing in the system depicted in Figure 5.11 are exactly the same as in the hardware implementation: two BRAMs that store the program data and instructions, respectively; an additional *OPB Timer* that is used to measure the time elapsed until the end of a scheduling process, and a *RS232 UART* to establish a serial communication with the external PC.

5.2.2. Software scheduler: Low-level implementation details

The pseudo-code of this scheduler is depicted in Figure 5.12. It works as follows: First of all, the *OPB Timers* and the *Interrupt Controller* are initialized (Line 25) and the interrupts (Line 26) are configured in such a way that each counter is assigned a different *Interrupt Handling Routine* (IHR in the figure). Then, it initializes the data structures (Line 27): the task graph, the array of events (initially with a *new_task_graph* event), the array of reconfigurations and the reconfigurable units, which are initially empty (Lines 1-3). And finally, it performs the scheduling process, which is divided into the design-time phase (Line 28) and the run-time phase (Line 30).

Since the purpose of this environment is to evaluate the performance of the run-time scheduler, the additional OPB timer only has to measure the elapsed time during the run-time phase. Hence, the first measure of the clock count is performed immediately before this phase (Line 29).

```
1. Task_graph: array [1..NUM_NODES] of Node;
2. Events: array [1..MAX_NUM_EVENTS] of Event;
3. RUs: array [1..NUM_RUs] of RU;
4. void code_run_time_scheduler (){
5.   initialize_scheduler ();
6.   WHILE (not finished){
7.     IF (Events not empty){
8.       Event ev := extract_event (&Events);
9.       SWITCH (ev){
10.        case 0: new_task_graph_event ();
11.        case 1: end_of_reconfiguration_event (ev.source);
12.        case 2: end_execution (ev.source);
13. }}}}
14. void IHR_0 (){ code_IHR (0); }            //IHRs of the counters
...
15. void IHR_N (){ code_IHR (n); }
16. void code_IHR (int n){
17.   stop_counter (n);                       //Stops simulating Counter n
18.   IF (counter_activated (n) == 0){        //Counter 0 interrupted
19.     generate_event_end_reconfiguration ();
20.   }ELSE{                                  //Counter 1 interrupted
21.     generate_event_end_execution ();
22.   }
23. }
24. int main (){                            //MAIN PROGRAM
25.   initialize_peripherals ();
26.   configure_interrupts ();
27.   initialize_data_structures ();
28.   design_time_scheduler ();
29.   int t1 = getValue_timer (&timer);
30.   code_run_time_scheduler ();
31.   int t2 = getValue_timer (&timer);
32.   printf (t2-t1);                  //Shows the elapsed time
33. }
```

Figure 5.12. Pseudo-code of the software implementation of the scheduler

The run-time scheduler is executed next, which main loop (Lines 6-13) iterates until the system has generated *NUM_NODES* times an *end_of_execution* event, where *NUM_NODES* is the number of nodes in the task graph. When this happens, the *finished* variable (Line 6) is activated. This does not appear in the figure for simplicity.

Whenever the scheduler has to send a command to start a reconfiguration or an execution for a task in a reconfigurable unit, it actually sends a command to count a specific number of clock cycles to the corresponding timer, according to

the reconfiguration or execution times of the involved task, and the operating frequency of the bus. The desired latency is simulated this way. When the counter finishes, it generates an interrupt, which triggers the execution of the corresponding interrupt handling routine (Lines 14-15). This allows knowing which reconfigurable unit has generated the interrupt. Next, the routine identifies which counter has interrupted (either the counter for the reconfigurations or the one for executions, Lines 18-21) and generates the proper event. When this management finishes, the execution flow switches to the main loop again (Line 6). It automatically detects that a new event has been generated (Line 7), which triggers the proper scheduling actions (described in the previous chapter) and conveniently updates the data structures. This is performed in the functions *new_task_graph_event()*, *end_of_reconfiguration_event() and end_of_execution _event()* (Lines 10-12).

When the main loop identifies that it has handled *NUM_NODES* times the *end_of_execution* event, it finishes (Line 6). Finally, back in the *main()* function, the value of the additional timer is retrieved and compared with the previous one (Lines 31-32) in order to know the execution time of the task graph in the software scheduler.

5.3. Conclusions

This chapter has described in detail the two implementations of our run-time task-graph scheduler. First of all, our hardware implementation of the scheduler has been described, which consists in a hardware module that uses some of the reconfigurable resources existing in the FPGA. The equivalent software version is presented next, which is a program that runs in one of the embedded Power PC microprocessors existing in the Virtex-II Pro.

As shown in this chapter, the hardware implementation of the scheduler comprises several modules to store the necessary information (a table for the task graphs and two queues for the sequence of reconfigurations and the run-time events, respectively), several modules that simulate the behaviour of the reconfigurable units, a replacement module that implements the LF+C replacement technique, and a control unit that manages the overall operation of the system.

This hardware module has been integrated as a peripheral in a microprocessor-based system in order to test its run-time performance. This customized peripheral implements the bus communication interface in order to capture the information about the task graphs to be executed that the processor sends to the scheduler. In addition, this system includes a programmable counter that is able to measure with clock-cycle accuracy the run-time delays that this hardware scheduler generates. These experimental results will be analysed in the following chapter.

On the other hand, the equivalent software version of the scheduler is a software program that runs in one of the embedded Power PC microprocessors existing in the Virtex-II Pro and that communicates with a set of programmable counters, which simulate the reconfiguration and execution latencies of the tasks in the reconfigurable units. The scheduling computations are performed in the Power PC, which communicates with these programmable counters by means of

an event generation and handling mechanism. Thus, each time the scheduler triggers the reconfiguration or an execution of a task in the system, the Power PC sends a request to the appropriate counter indicating the number of clock cycles corresponding to the involved latency. When this count finishes, the involved counter generates an interrupt, which is interpreted by the scheduler as a run-time event. This event is finally handled by the software running in the embedded processor.

These two implementations are intended to offer different trade-offs in terms of resources consumptions and run-time delays that the scheduler generates. The hardware version has been designed to provide a high performance at the cost of consuming a low amount of reconfigurable resources, whereas the software one does not have any hardware implementation cost, but it introduces more run-time delays. This point will be deeply analysed in the next chapter.

Never regret anything that made you, or another, smile

Anonymous

Chapter 6: Experimental results

This chapter presents the experimental results obtained throughout this doctoral thesis. Thus, Section 6.1 presents the synthesis results of the hardware implementation, and compares them with those of the equivalent software version. Then, Section 6.2 evaluates and compares the run-time overheads generated by both implementations. The comparative results of these two sections will show that both implementations offer two different trade-offs between hardware resources consumption and performance, thereby making them suitable for different scenarios. Section 6.3 evaluates the performance of the proposed scheduling technique and compares it with other well-known scheduling techniques, showing the benefits of our approach. Finally, Section 6.4 summarizes this chapter with some final conclusions.

6.1. Synthesis results

This section makes a comprehensive analysis of the resources consumption and the operating frequency of both the hardware and the software versions of the scheduler in our Virtex-II Pro XC2VP30. First of all, the scheduler itself is evaluated and then, the section gives more details about the microprocessor-based system in which the scheduler has been integrated for testing and evaluation purposes.

Figure 6.1 shows the resources consumption for a scheduler that includes different numbers of reconfigurable units. The figure shows the implementation cost from 3 to 8 reconfigurable units, although it is fully scalable as long as there are enough resources to implement it. In each case, the figure shows the consumption of five kinds of resources: slices, slice flip flops, 4-input LUTs, Block RAMs and 18-bit-wide multipliers. The Virtex-II Pro used for these experiments includes 13969 slices, 136 BRAMs and 136 multipliers. Each slice contains two slice flip-flops and two 4-input LUTs.

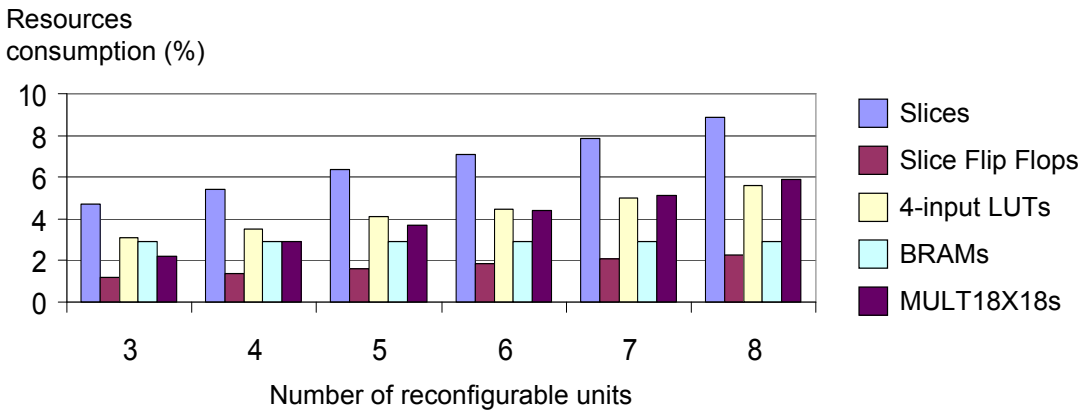

Figure 6.1. Implementation cost for the hardware scheduler with different number of reconfigurable units

As shown in the figure, the implementation cost of the scheduler increases as the number of reconfigurable units grows. This is due to the growing number of modules that the scheduler has to include in order to manage the reconfigurable slots, as well as to the complexity of the control unit and the volume of existing communications. The percentages of all the types of resources increase except the number of BRAMs, since the scheduler only uses four of them (as it will be shown later): two for the task-graph table and one for the reconfigurations and events queue, respectively, which is a 3% of the total number of BRAMs in the FPGA. In any case, the cost is always very affordable and none of the percentages grows to above 9% of each class of the available resources.

As commented in the previous chapter, the proposed implementation of the scheduler has more customizable parameters: the size of the task-graph table, as well as the sizes of the reconfigurations and event queues. However, according to our experiments, the resources consumption of the scheduler barely changes as these parameters vary: the difference between the worst and best cases is less than 1%. The reason is that the information about the task graph, the reconfigurations and the event queues is always stored in BRAMs (which size is always constant); hence this only has impact on the complexity of the controllers of the respective modules. As for the scalability of the scheduler regarding these parameters, these modules can have up to 512 entries, which is plenty of space to store all the information needed in order to carry out the experiments shown throughout this chapter.

We have also evaluated the resources consumption for each one of the modules the scheduler is composed of. Table 6.1 shows the detailed results. The most expensive module is the control unit, which uses 4.34% of the slices of the FPGA. However, this consumption is very affordable for this size (eight reconfigurable units).

Table 6.1. Detailed implementation cost for the hardware scheduler with eight reconfigurable units

Module	# of slices (%)	# of Slice Flip Flops (%)	# of 4-input LUTs (%)	# of BRAMs (%)	# of multipliers (%)
Task Graph Table	157 (*1.15%*)	74 (*0.2%*)	278 (*0.74%*)	2 (*1.47%*)	0 (*0%*)
Reconfigurations queue	9 (*0.07%*)	10 (*0.03%*)	20 (*0.05%*)	1 (*0.74%*)	0 (*0%*)
Events queue	13 (*0.09%*)	13 (*0.03%*)	27 (*0.07%*)	1 (*0.74%*)	0 (*0%*)
Module for the RUs	55 (*0.4%*)	81 (*0.22%*)	94 (*0.25%*)	0 (*0%*)	1 (*0.74%*)
Control unit	595 (*4.34%*)	106 (*0.28%*)	1024 (*2.74%*)	0 (*0%*)	0 (*0%*)

As in Figure 6.1, Figure 6.2 shows the resources consumption for the microprocessor-based system in which the scheduler has been integrated and that was described in the previous chapter. The figure shows what happens when the scheduler includes a variable number of reconfigurable units that range from 3 to 8. The variation in the number of reconfigurable units does not have a great impact on the resources consumption of the total microprocessor-based system. For instance, the increment between the worst and the best cases is 4% for the number of slices and barely 1% for the number of flip flops. This is coherent with the results obtained in Figure 6.1. Comparing Figures 6.1 and 6.2 we can also appreciate the additional implementation cost that the microprocessor-based system itself introduces.

The operating frequency of the scheduler is approximately 103 MHz and it barely changes when the number of reconfigurable units grows. Hence, it has been possible to integrate it in a system with EDK in which the processor, the buses and all the peripherals work at 100 MHz.

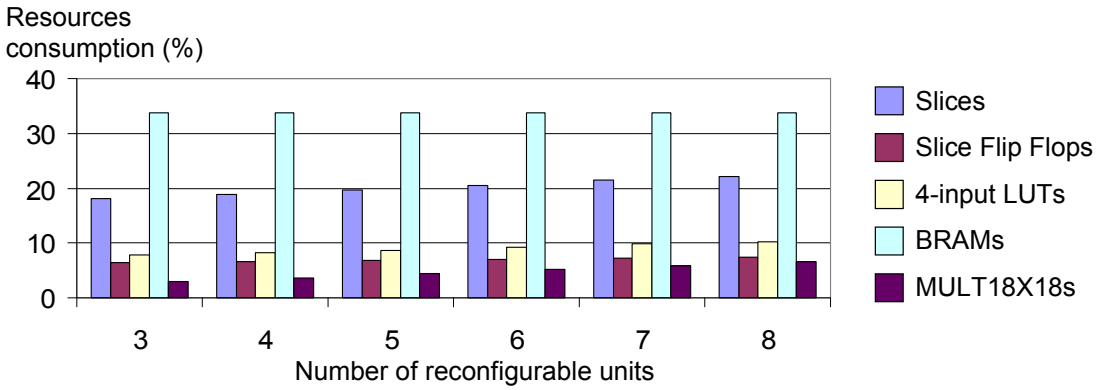

Figure 6.2. Implementation cost for a microprocessor-based system with a scheduler with different number of reconfigurable units

Finally, I will finish this section comparing the size of the codes of the software programs that run in the microprocessor-based systems that include the hardware and the software implementations of the scheduler. In embedded systems this is also an important metric since the memory space is normally very restricted. For instance, a Virtex-II Pro XC2VP30 FPGA only has 136 Block RAMs, (each one with a size of just 2.25 KB) and 53.5 KB of distributed RAM which makes a total of 359.5 KB of RAM memory. This distributed RAM is spread in all the existing logical blocks of the FPGA. The size of the *.elf* file of the software implementation of the scheduler is 118 KB, whereas for the hardware implementation it is 85.5 KB. These sizes include not only the scheduler, but also all the drivers for the peripherals included in the system. Hence, although the hardware implementation of our scheduler has some hardware area cost, it reduces the system memory requirements. This can be especially useful to reduce the accesses to external memories, for instance.

6.2. Run-time overhead generated by the scheduler

The microprocessor-based systems described in the previous chapter were intended to measure the run-time delay generated by the hardware and software implementations of the scheduler, respectively. This section shows these experimental results and makes a comparison between both versions.

The experiments carried out in this section (and in the following one) evaluate the scheduler using a set of task graphs extracted from actual multimedia applications: two versions of a JPEG decoder (JPEG and Parallel-JPEG), a MPEG-1 encoder, a pattern recognition application (that applies the Hough transform over a matrix of pixels to identify geometrical figures), and a 3D rendering application based on the open source Pocket-GL library (Pocket GL) [RMC05, RMVC05]. In the latter case the application includes 20 different task graphs that, for simplicity, have been grouped in four categories (from A to D, which contain 2, 5, 9 and 4 tasks graphs, respectively), taking into account their number of tasks (2, 4, 5 and 6 tasks, respectively).

Table 6.2 presents the delays that the software scheduler introduces for these task graphs. Column 2 shows their initial execution time assuming that the scheduler does not introduce any delay, whereas Columns 3–6 provide details regarding the overhead generated by the software scheduler. Thus, Column 3 shows the total execution times and Columns 4 and 5 break down the delays introduced by the software implementation of the scheduler into computation and communication times, respectively. The computation times include the execution time of the code that deals with the task-graph scheduling, whereas the communication times include the interrupt handling (which in turn implies switching into the corresponding interrupt service routine, stopping/resuming the timer and enabling/disabling its interrupts) and the initialization of the reconfigurable units. Column 6 presents the impact of these overheads with

respect to the initial execution time. Thus, *Column 6 = (Column 3 – Column 2) / Column 2 * 100*.

Table 6.2. Performance evaluation of the software task-graph scheduler

Task graph	Initial execution time (ms)	Performance of the software scheduler			
		Execution time (ms)	Management time (ms)		Overhead (%)
			Compu-tations	Commu-nications	
JPEG	79	80.07	0.771	0.298	1.35
PARALLEL-JPEG	54	55.44	0.872	0.563	2.67
MPEG-1	37	38.27	0.915	0.358	3.43
HOUGH	94	95.08	0.729	0.355	1.15
POCKET GL (A)	4.10	5.25	0.968	0.179	28.05
POCKET GL (B)	16.02	17.60	1.279	0.298	9.86
POCKET GL (C)	26.89	28.56	1.316	0.358	6.21
POCKET GL (D)	48.75	50.50	1.336	0.417	3.59

Table 6.3 shows the delays that the hardware scheduler generates for the evaluated task graphs. As in Table 6.2, Column 2 shows their initial execution time assuming that the scheduler does not introduce any delay, whereas Columns 3 and 4 provide details regarding the overhead generated by the hardware scheduler. In this case, the delays shown in Column 3 are introduced at run time by the logic of the scheduler and the hardware/software communications among the processor and the reconfigurable units. On the one hand, the delays introduced due to the computations are just a few hundreds of cycles in a system running at 100 MHz (i.e. a few microseconds). On the other hand, the communication delays are approximately 2200 clock cycles when using a DMA. The reason of this drastic reduction of these communication delays is that the amount of interrupts that the processor must handle is drastically reduced in this case, since the event management is carried out internally in the hardware block and the only interrupt that it generates is raised at the end of

each scheduling process.

Finally, Column 4 presents the impact of the overheads of Column 3 with respect to the initial execution time of the tasks. Thus, *Column 4 = (Column 3 – Column 2) / Column 2 * 100*.

Table 6.3. Performance evaluation of the hardware task-graph scheduler

Task graph	Initial execution time (ms)	Performance of the hardware scheduler	
		Execution time (ms)	Overhead (%)
JPEG	79	79.022	0.03
PARALLEL-JPEG	54	54.023	0.04
MPEG-1	37	37.023	0.06
HOUGH	94	94.022	0.02
POCKET GL (A)	4.10	4.122	0.54
POCKET GL (B)	16.02	16.042	0.14
POCKET GL (C)	26.89	26.912	0.08
POCKET GL (D)	48.75	48.772	0.05

As Table 6.2 shows, the software implementation generates delays that go from 1% to 28% of the initial execution time. For applications with high execution times (such as HOUGH, for instance) these overheads are not especially meaningful. However, there are some cases (in applications with small execution times, such as POCKET-GL) in which this delay can generate a very important penalty in the performance of the application (up to 28%), since this overhead is quite significant with respect to its execution time. In the latter cases these delays are not acceptable, and both communication and management times can be reduced by using the hardware implementation, as Table 6.3 illustrates. Thus, communication times are reduced on average from 0.35 milliseconds to just 0.022 milliseconds (i.e. by 1-2 orders of magnitude), and the computation times

are also drastically reduced (from 1.02 milliseconds to 0.002 milliseconds on average, which is between 2 and 3 orders of magnitude lower).

6.3. Performance evaluation

This section evaluates the performance of the proposed scheduler. First of all, Subsection 6.3.1 shows the benefits of the proposed scheduler when executing just one task graph. Then, Subsection 6.3.2 does likewise when the scheduler executes two task graphs following a defined execution pattern, and compares the proposed LF+C replacement technique with other well-known replacement policies, such as LRU and LFD. It is interesting to evaluate these points because these applications are normally cyclic and/or execute several consecutive times. Subsection 6.3.3 shows the benefits of taking into account the mobility during the scheduling process in order to delay some of the reconfigurations (*Skip-Events* approach) and finally, Subsection 6.3.4 gives performance results in terms of final execution times of the involved applications.

6.3.1. Impact of the prefetch and reuse techniques

Tables 6.4 and 6.5 present detailed information about the evaluated task graphs, which are divided into two different groups. Group 1 includes the JPEG, Parallel-JPEG, MPEG-1 and HOUGH applications, whereas Group 2 includes the task graphs from the Pocket-GL application. As in Tables 6.2 and 6.3, the tasks belonging to Group 2 are divided into categories A-D depending on their number of tasks.

For each task graph, both tables show its number of tasks (Column 2) and their initial execution time (Column 3), which represents an ideal scenario with no delays due to the run-time management or due to the reconfigurations. Finally, Column 4 of both tables shows the delays that are generated due to the reconfiguration overheads when using an on-demand approach; i.e. the reconfigurations start when the tasks must be executed. We assume that the

reconfiguration latency is 4 milliseconds, which is the time needed to reconfigure the largest task used with a reconfiguration clock frequency of 50 MHz. As the table shows, the reconfiguration overheads generated by an on-demand approach is very important, especially for the Pocket GL application, where it can even be greater than its initial execution time.

Table 6.4. Details of the task graphs used as benchmarks in the performance evaluation belonging to Group 1

Task graphs Group 1			
Task graph	**Number of tasks**	**Initial execution time (ms)**	**On-demand reconfiguration overhead (ms)**
JPEG	4	79	16
Parallel-JPEG	8	54	32
MPEG-1	5	37	20
HOUGH	6	94	24

Table 6.5. Details of the task graphs used as benchmarks in the performance evaluation belonging to Group 2

Task graphs Group 2			
Task graph	**Number of tasks**	**Initial execution time (ms)**	**On-demand reconfiguration overhead (ms)**
POCKET GL (A)	2	4.10	8
POCKET GL (B)	4	16.02	16
POCKET GL (C)	5	26.89	20
POCKET GL (D)	6	48.75	24

In order to evaluate the impact of the prefetch and reuse techniques, the task graphs from Tables 6.4 and 6.5 have been evaluated separately by executing them several consecutive times on a system that includes a variable number of reconfigurable units. Figure 6.3 shows the reduction in the reconfiguration overhead due to these optimizations.

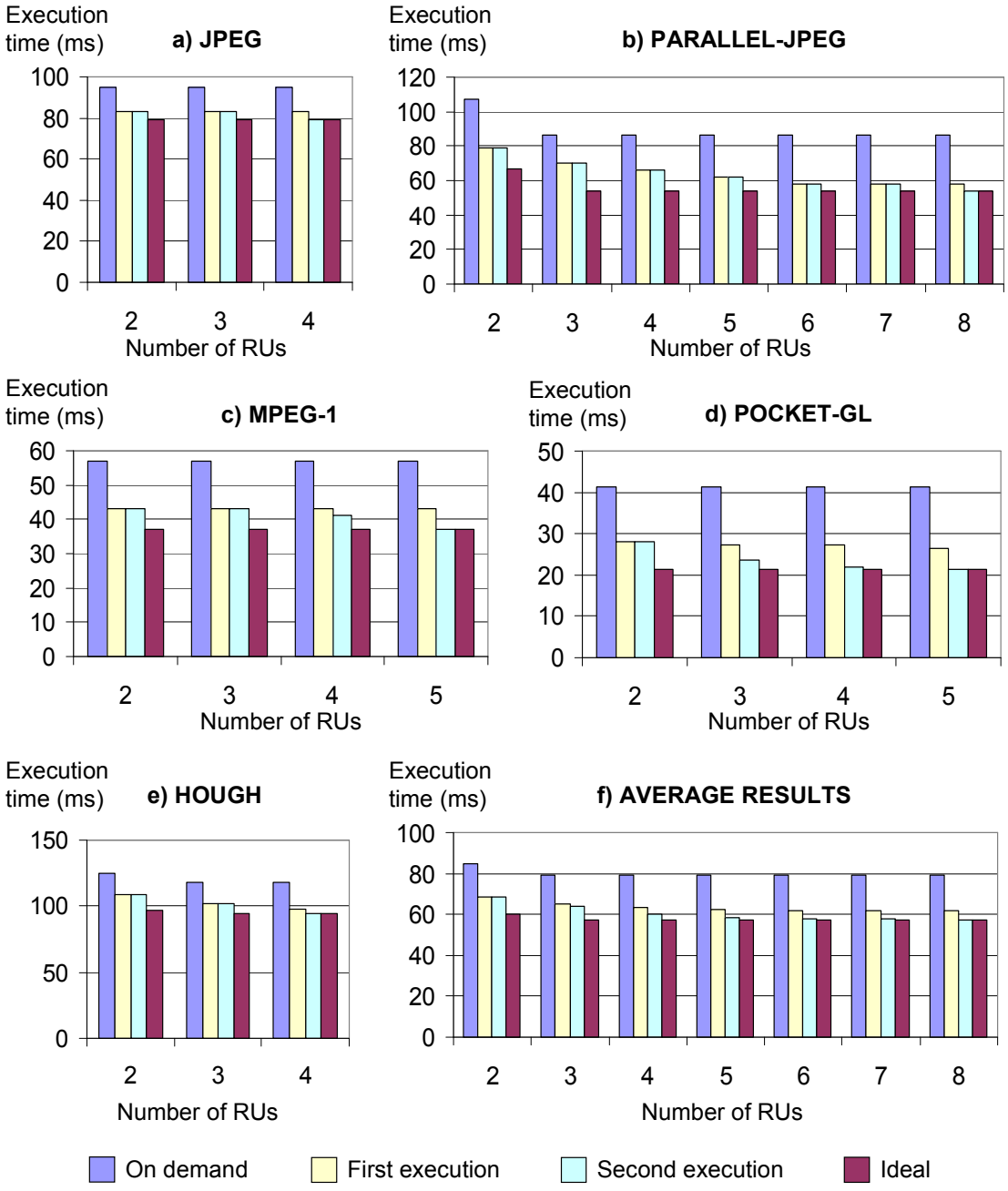

Figure 6.3. Benefits of applying the prefetch and reuse optimizations during the scheduling process

This figure shows four different execution times:

- **On demand**: It represents a situation in which no optimization is performed. Hence, none of the reconfigurations is prefetched and all of them generate temporal overheads.
- **First execution**: In this case, the prefetch technique is applied when the task graphs are executed once. This experiment does not show the impact of applying the LF+C replacement technique, since when the system executes just one task graph the scheduler cannot reuse any task from a previous execution.
- **Second execution**: In this case, both the prefetch and reuse techniques are applied when the task graphs are executed two consecutive times. The execution time that is shown corresponds to the second execution and allows evaluating the additional benefits of our LF+C replacement technique.
- **Ideal**: It represents the execution time for the task graphs without any reconfiguration overhead. This is an ideal situation and actually represents an upper-bound in terms of performance for the scheduler.

In order to evaluate the impact of just the basic prefetch and reuse techniques, in these experiments the mobility of the tasks has been set to zero; hence in this case the *Skip-Events* feature is not applied. Figure 6.3 shows what happens for different numbers of reconfigurable units.

As the figure shows, the on-demand strategy is very inefficient, and the prefetch optimization can greatly improve the system performance. On average, the reconfigurations generate a 37% execution time overhead when using the *on-demand* approach, whereas it only generates about 10% overhead when applying the prefetch optimization. In addition, this overhead is reduced to just 5.6% when the manager can reuse some of the tasks.

6.3.2. Benefits of using the LF+C replacement policy

This subsection evaluates the benefits of applying LF+C rather than other replacement techniques. For this purpose, we have carried out a set of experiments that consist in the execution of two task graphs in our scheduler following an execution pattern in which both tasks are executed alternatively. For instance, if the two task graphs involved are JPEG and MPEG-1, the execution pattern would be: JPEG – MPEG-1 – JPEG – MPEG-1…. Each combination has been executed several iterations and the performance of the scheduler has been evaluated without taking into account the first one, because during the first execution it is not possible to reuse anything. As in Table 6.3, which divides the evaluated task graphs into two groups (1 and 2); the results shown in this subsection evaluate separately the tasks belonging to Group 1 and Group 2, respectively.

For the tasks belonging to Group 1, we have evaluated all the possible execution patterns that result from selecting two task graphs from the total set of four. Thus, since there are four possible tasks in Group 1 (JPEG, MPEG-1, PARALLEL-JPEG and HOUGH), the number of possible combinations of two elements taken from this set, allowing repeated elements and caring about their order is 16. Hence, a total number of 16 execution patterns have been evaluated for the task graphs belonging to Group 1. On average, each set of two task graphs contains 10.06 tasks, which have been executed in such a way that there are always significantly more active tasks than reconfigurable units. These results are particularly interesting since these experiments are so demanding that the only way to obtain a good performance is to apply an efficient replacement policy. More specifically, for each one of these patterns, we have evaluated what happens when the number of reconfigurable units ranges from 3 to 9 (which involve 7 experiments per execution pattern). Hence, a total number of *16*7=112* experiments have been carried out.

Figure 6.4 shows the average results, which are labelled as *"Remaining Rec. Overhead %*" and "*Reuse %*" in Figure 6.4.a and Figure 6.4.b, respectively. The *remaining reconfiguration overhead percentage* is defined as the percentage of the original reconfiguration overhead that remains after applying the scheduling techniques, whereas the *reuse percentage* is defined as the number of reused tasks divided by the total number of executed tasks.

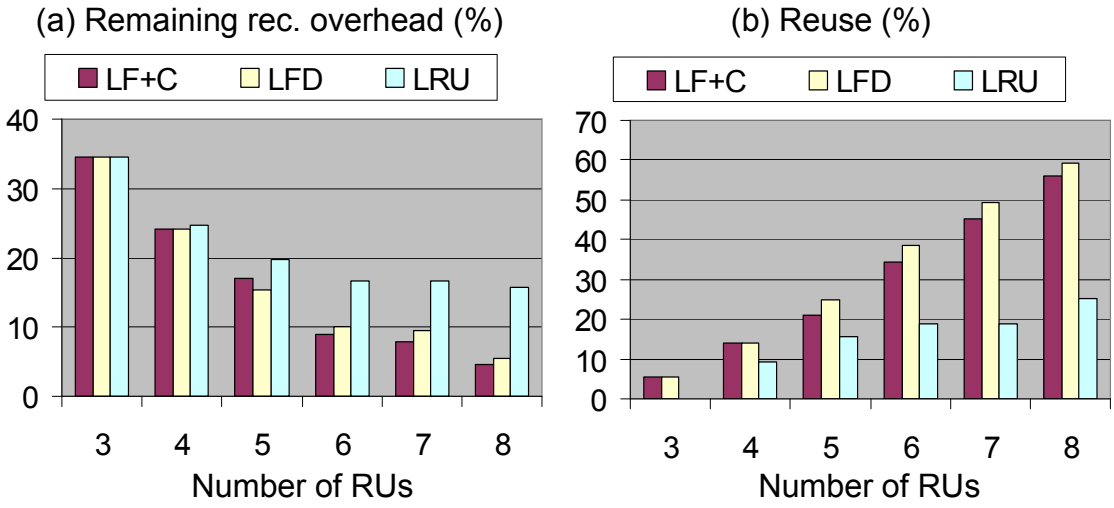

Figure 6.4. Evaluation of the execution of the task graphs in Group 1 with different replacement policies

In these experiments, the system applies our scheduling algorithm without delaying any reconfiguration (a purely ASAP approach, assuming that the mobility of all the tasks is 0). It then compares the LF+C replacement strategy with two other well-known replacement heuristics: LRU and LFD.

As Figure 6.4.a shows, the prefetch technique is a powerful means to reduce the reconfiguration overheads, since it can remove most of the original overhead even when no significant reuse is achieved (for instance, for three RUs using LRU, no task is reused and the scheduler manages to hide 65% of the original reconfiguration overhead). However, the remaining overhead is still important

(35%). Hence, there is still much room for improvement, which can be achieved thanks to the inter-task reuse.

It is clear that as more reconfigurable units are included in the system, the reuse percentage increases and this helps to reduce the overhead for the three replacement policies. Nevertheless, the results obtained when using a LRU approach are clearly suboptimal: always above 15% of the remaining reconfiguration overhead and reusing less than 25% of the involved tasks. However, LFD reduces the reconfiguration overheads and greatly increases the reuse percentage, especially as the number of reconfigurable units grows. Since LFD guarantees the optimal results for the latter problem, these reuse percentages can be used as the upper-bound that a replacement policy can achieve. It is also important to remind that LFD obtains the optimal results regarding reuse, but it can only be applied to static systems when all the future events are known, whereas LF+C has been designed to be applied at run time, and does not need any information about the future task graphs to be executed.

Finally LF+C obtains almost as good results regarding reuse as LFD, and in some cases even better performance. In fact, when the number of RUs is greater than 5, by using LF+C, the scheduler generates 1.17% less reconfiguration overhead than LFD, and it reuses 3.82% fewer tasks. The reason is that although LF+C reuses a slightly smaller percentage of tasks than LFD, most of these tasks are critical (since LF+C assigns them more priority), and reusing critical tasks has a direct impact in the performance of the system.

For the tasks belonging to Group 2 (the Pocket GL application), we have simulated 500 iterations and in each one of them, one of the 20 task graphs has been randomly selected from the Pocket-GL library. Although there are many different task graphs, reusing tasks is possible in this experiment because all of them include some common tasks; in fact there are just 10 different tasks in these 20 task graphs. Figure 6.5 shows the results for this experiment, which confirm that the LF+C replacement policy achieves almost optimal results regarding reuse (on average, just 3% worse than LFD) and very good results

regarding performance (on average, just 0.65% worse than LFD. However, the comparison is favourable for LF+C and for 7 and 8 RUs). The figure also shows that the difference between LF+C and LRU is again very significant, even though the system uses more reconfigurable units. For instance, for 8 RUs, if the system uses LF+C the overhead almost disappears, whereas if it uses the LRU approach, a percentage of almost 10% still penalizes the system performance.

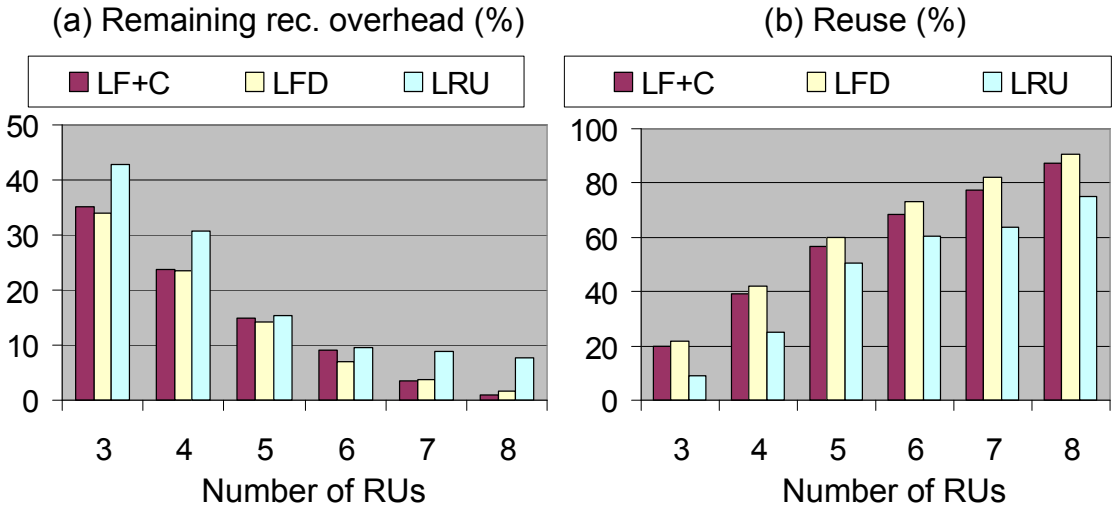

Figure 6.5. Evaluation of the execution of the task graphs in Group 2 with different replacement policies

Finally, another effect that can be observed in the plots in Figure 6.5 is the smaller gap in the percentages of performance and reuse between LF+C and LRU with respect to Figure 6.4. The reason is that it is much more difficult to obtain good scheduling results in the second experiment due to the unpredictable behaviour that the random selection of the task graphs introduces. In any case, these two experiments show that LF+C outperforms LRU on average, while obtaining very competitive results with respect to LFD, even though the scheduler still does not include the *Skip-Events* feature, which is a great result.

6.3.3. Comparison between the *Skip-Events* and ASAP approaches

The results of the previous subsection have proved that the LF+C replacement policy obtains very competitive results in comparison with LFD regarding both average reconfiguration overhead reduction and task reuse. However, these results can be further improved if the scheduler takes into account the mobility of the tasks in order to delay some reconfigurations (*Skip-Events* approach). This subsection evaluates the impact of applying this technique.

The experiments carried out in this subsection are the same as in the previous one, but this time comparing LF+C with and without the *Skip-Events* feature. Figure 6.6 shows the performance evaluation for the tasks belonging to Group 1, whereas Figure 6.7 does likewise for the tasks belonging to Group 2.

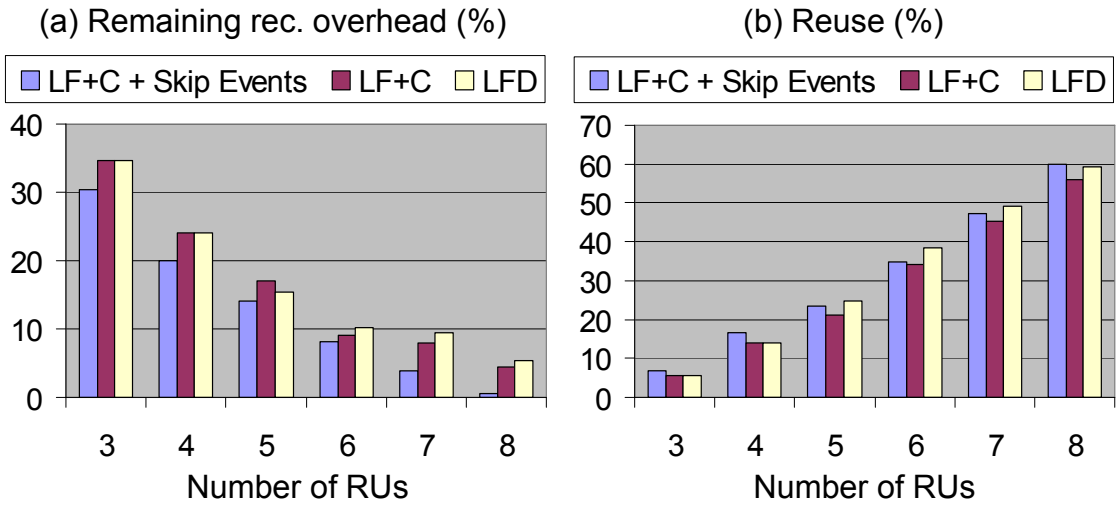

Figure 6.6. Comparison between the LFD, LF+C and LF+C + Skip Events replacement policies for the same experiment presented in Figure 6.4 (task graphs for Group 1)

On the one hand, both figures show that by delaying some reconfigurations the scheduler always improves the performance compared to the equivalent ASAP approach by 3.3% and 6.7% on average, for tasks belonging to Group 1 and Group 2, respectively. In addition, the reuse percentage improves by 2.2% and 4.5% for tasks belonging to Group 1 and Group 2, respectively. Moreover, in some particular cases the *Skip-Events* feature greatly outperforms the ASAP approach. For instance, for the experiment with the Pocket-GL library (Figure 6.7) with 5 reconfigurable units, this technique reduces the remaining reconfiguration overhead of LF+C by 11.01% on average, whereas the reuse rate is 7.94% higher.

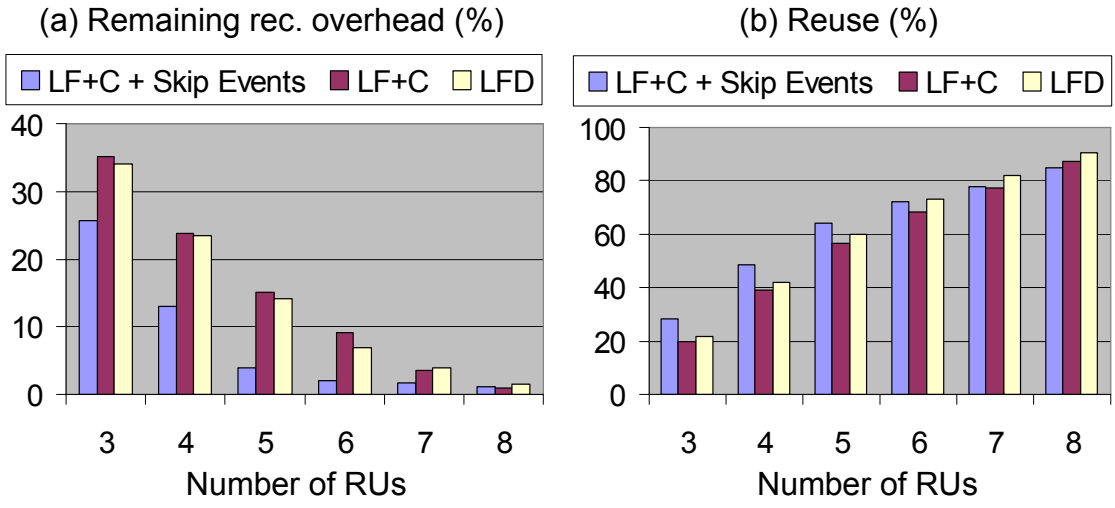

Figure 6.7. Comparison between the LFD, LF+C and LF+C + Skip Events replacement policies for the same experiment presented in Figure 6.5 (task graphs for Group 2)

On the other hand, if we compare LFD with our *LF+C + Skip Events* replacement technique, we can observe that the latter always outperforms LFD in terms of reconfiguration overhead reduction. In fact, according to the results displayed in Figure 6.6.a, and in Figure 6.7.a, the remaining reconfiguration overhead is reduced by 3.62% and by 6.06% on average, respectively. Moreover, according to Figures 6.6.b and 6.7.b, in some cases *LF+C + Skip Events* obtains better reuse rates than LFD. In fact, this happens for 3, 4 and 8

RUs in Figure 6.6.b and for 3, 4 and 5 RUs in Figure 6.7b. These results may look strange, since we are claiming to achieve “better results than the optimal ones”. Clearly the reason is that we are playing with different rules, because LFD cannot delay any reconfiguration, whereas our proposed replacement policy takes advantage of this optimization opportunity.

In any case, the results of these two figures show how the trend of the previous subsection becomes even more accentuated, in which our replacement technique reuses more tasks and clearly outperforms the LRU and LFD replacement techniques.

6.3.4. Execution time evaluation

Finally, Figure 6.8 shows the results of our experiments regarding execution time. Figure 6.8.a shows the average execution times for the task graphs of Group 1, whereas Figure 6.8.b refers to Group 2. In both cases, the figure compares the results with the *optimal execution times* that have been obtained using our scheduler, but assuming that there is not any reconfiguration overhead. Hence, these schedules represent an upper bound in terms of performance for the final schedule, which are unachievable in many cases (especially when the number of reconfigurable units is small).

These results demonstrate that our scheduler provides near-optimal results regarding the reduction of the reconfiguration overheads as long as enough reconfigurable units are available to take advantage of the LF+C replacement policy. For instance, when the system includes more than four reconfigurable units, the difference between the upper-bound and the actual results is always less than three milliseconds in both cases, and it almost disappears when there are more than 7 units. The average number of tasks executed in the experiments of Figure 6.8 is approximately 10. Hence the developed scheduler obtains near-

optimal results even when the number of task almost doubles the number of reconfigurable units.

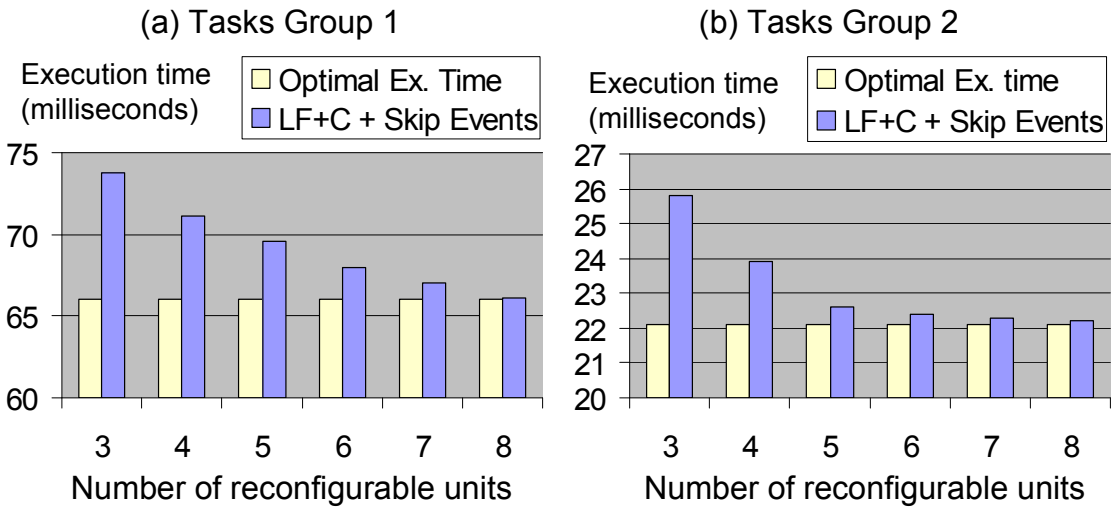

Figure 6.8. Comparison of the average execution times in our experiments with the optimal execution times for the tasks belonging to Group 1 (a) and Group 2 (b)

6.4. Conclusions

This chapter has evaluated the proposed scheduler from several points of view and has compared the hardware and software implementations that have been developed in this doctoral thesis.

The first section of the chapter has analysed the resources needed to implement the proposed scheduler. Our experiments have shown that the implementation cost for the hardware version is very affordable, since it uses less than 9% of the available slices, Block RAMs and multiplicators in our small FPGA. The microprocessor-based system built to test the scheduler does not introduce an additional meaningful cost either, since the whole system consumes approximately 13% more slices than the scheduler itself, and 40 more BRAMs, which suppose an additional 28% of the BRAMs existing in our Virtex-II Pro FPGA. Obviously, the software version does not consume any hardware resources; however the size of the code that has to be executed in the Power PC and that actually implements the software scheduler is approximately 30% bigger.

Next, the run-time delays generated by both implementations of the scheduler have been evaluated and compared. Our experiments have shown the convenience of using the hardware implementation if a good performance is required, since it reduces by 2-3 orders of magnitude the run-time penalty that the software implementation generates. This reduction is very convenient in many cases; for instance, the run-time penalty generated by the software scheduler can be up to 28% of the initial execution time for some of the task graphs. Hence, considering at the same time these run-time penalties and the implementation cost of both versions of the scheduler, we can observe that both of them offer different area/performance trade-offs. Hence either of them can be selected depending on whether the designer prefers to maximize the performance of minimize the area cost.

The performance of the scheduler has been finally evaluated. For that purpose, the benefits of the prefetch and reuse techniques have been measured first. Both of them have proved to be very efficient, since they reduce the reconfiguration overhead from 37% (which corresponds to an on-demand approach) to just 5.6% on average when the same task graph is executed several consecutive times.

The last two sections of this chapter have shown the experiments carried out in order to test the efficiency of our LF+C replacement technique. The results have shown that it outperforms other well-known replacement techniques, such as LFD and LRU. LFD is the one that provides the optimal reuse rate and can only be applied for design-time problems, since it needs to know all the tasks that are going to be executed in the future. Hence it cannot be applied in dynamic systems. However, it can be used to obtain an upper-bound value for the reuse. This is done recording all the events generated during the execution of a given experiment and afterwards, repeating the same experiment but using LFD.

The fact that our replacement strategy achieves better results than LFD *regarding performance* (i.e. in terms of reconfiguration overhead reductions) is very meaningful, since this demonstrates that in order to achieve good overhead reductions, it is not only necessary to reuse many tasks, but to reuse wisely. This is possible because LF+C takes into account the criticality of the potential victims for replacement and avoids replacing them as far as possible. And this is achieved in spite that LFD knows all the task graphs that are going to be executed in the future whereas LF+C does not have this information, which makes the obtained results even more brilliant. Finally, the last experiments have shown that the *Skip-Events* feature clearly outperforms an equivalent ASAP approach when combined with the LF+C replacement policy, in terms of reconfiguration overhead, execution time reductions and task reuse. In addition, our results have shown that the *Skip-Events* feature allows our scheduler to obtain better reuse rates than LFD in some cases. The reason of these nice results is that LFD cannot delay any reconfiguration, whereas the proposed replacement policy takes advantage of this optimization opportunity.

If you're not prepared to be wrong, you'll never come up with anything original

Ken Robinson

Chapter 7: Conclusions and future work

Many current applications, such as signal and image processing, and vision applications have become more and more computationally intensive in the last few years. For this reason, embedded execution platforms have evolved into complex Systems-On-Chip, many of which usually include ASICs. These are integrated circuits customized for a particular use that are designed to achieve optimal or near-optimal performance. However, they usually lack the flexibility needed to adapt the platform to different execution contexts in dynamic scenarios. In this sense, dynamically reconfigurable hardware has appeared as a promising technology that offers an interesting trade-off between flexibility and performance, which many recent embedded system applications demand.

However, one of the main drawbacks of dynamically reconfigurable hardware is the large latency related to the reconfiguration process, which is usually of the

order of hundreds of milliseconds. Hence if the reconfigurations are carried out very frequently, they can greatly degrade the performance of the running applications. This is the reason many techniques have been proposed to overcome this problem, such as reducing the size of the bitstreams, reducing the number of reconfigurations and designing the scheduling techniques for the applications taking into account the impact of their dynamic reconfigurations. The work developed in this doctoral thesis falls into the latter group of techniques.

As mentioned above, most modern embedded systems are characterized by their high degree of dynamism. In other words, at a given moment of time, the complete sequence of applications that will be executed in the future is partially or totally unknown. Hence, at design time it is usually impossible to know which sequence of tasks will be executed in the reconfigurable resources. Hence in order to achieve good scheduling decisions, a part of the scheduling process must be carried out at run time. However, if the scheduler performs many computations at run time, they can introduce important penalties in the system execution. Hence, these computations must be optimised in order to avoid further performance degradations.

In this sense, in the work developed in this doctoral thesis I propose a hybrid design-time/run-time task scheduler, which combines the best features of both purely design-time and run-time approaches. On the one hand, the decisions that are made at run time allow the system to adapt to dynamic and unexpected events. And on the other hand, performing the bulk of the computations at design time alleviates the run-time computational load of the system and prevents further performance degradations. Thus, the pursued goal of the proposed scheduler is to obtain high-quality schedules, as well as to speed up the computations that this scheduling process involves.

7.1. Contributions of this doctoral thesis

More specifically, the contributions of this doctoral thesis can be summarized as follows:

The first important contribution is the development of a **hybrid design-time/run-time scheduling algorithm** for applications that are executed in a hardware multi-tasking system comprised of a set of equal-sized reconfigurable units. These applications are represented as *Directed Acyclic Graphs*, which nodes represent computational tasks and which edges represent data dependencies among them. We assume that each node includes an estimation of its execution time. The involved task graphs are executed sequentially.

In a nutshell, the proposed scheduler firstly analyzes at design time the incoming task graphs in order to retrieve some useful information that will be used later at run time.

At design time, for each task graph, the scheduler characterizes all its tasks with three parameters: weight, criticality and mobility, which are used to obtain toe sequence of reconfigurations, to evaluate the impact of the reconfiguration of a task in the performance, and to identify whether it is possible to delay that reconfiguration.

At run time the scheduler basically follows an *As Soon As Possible* (ASAP) approach in order to load and execute the tasks in advance. For this purpose, the scheduler follows the sequence of reconfigurations that was obtained at design time. By following this ASAP approach, the scheduler hides the reconfiguration overhead of most of the tasks by overlapping their reconfigurations with the executions of other tasks. This technique is known in the literature as task prefetch. However, in certain situations the scheduler also delays some reconfigurations in order to avoid falling into sub-optimal solutions. For this purpose, the run-time scheduler uses the mobility parameter of the tasks.

In addition, I have also developed a replacement module and integrated it in the run-time scheduler in order to carry out the task replacements whenever a task must be reconfigured in the system. The replacement technique developed (which we have called *Look Forward* + *Critical* or LF+C) analyzes the available replacement candidates and decides at run time the replacement victim, giving more priority to those candidates that are critical and/or are going to be executed again in the near future. For this purpose, the replacement module uses the criticality parameter of the candidates.

According to the presented experiments, the developed scheduler hides up to 99% of the original reconfiguration overheads of the tasks in the recurrent execution of a set of task graphs extracted from actual multimedia applications. In addition, the LF+C replacement technique outperforms other well-known replacement policies, such as LRU and LFD, which is the one that achieves the optimal reuse rate. On the other hand, if the scheduler delays some of the reconfigurations depending on their mobility, the reuse rates achieved are even better than LFD.

The second important contribution of this doctoral thesis is **an efficient hardware implementation of the run-time scheduler** using a small amount of the reconfigurable hardware resources existing in the target FPGA, a Virtex-II Pro XC2VP30. The goal is to perform the run-time computations as fast as possible, thereby generating just a small run-time penalty (in this particular case, just a few clock cycles). In order to test this hardware module, a simplified simulation environment has been developed and implemented on the FPGA. This environment uses a set of programmable timers to simulate the behaviour of the reconfigurable units. This system also includes support to measure with clock-cycle precision the overheads generated by the scheduler. Hence it has been used to evaluate the performance of the scheduler.

In addition, this doctoral thesis makes a comparison of the hardware implementation of the run-time scheduler with an equivalent software one that consists in a program written in the C programming language and compiled for

the embedded Power PC microprocessors embedded in the used Virtex-II Pro. According to our results, the hardware version has proved to be up to three orders of magnitude faster than the software one. However, the latter has the advantage of not using any reconfigurable resources. Hence, these two versions offer different trade-offs between the run-time scheduling overhead and the cost needed to implement the hardware circuit.

7.2. Future work

There exist a number of interesting research lines to continue this work. Maybe the most straightforward one is to adapt the proposed scheduler for the execution of several tasks in parallel. The reason is that all the techniques described in this doctoral thesis assume that the task graphs are always sequentially executed. In this case, a natural extension of this work would be to develop a set of inter-task optimizations, such as analyzing the possibility of inter-task-graph prefetch and reuse, in order to further improve the schedules assigned to each involved task graph separately. In addition, another interesting research line complementary to this work would be to integrate the hardware implementation of the run-time scheduler in an actual hardware multi-tasking system based on partial reconfiguration. These extensions are not trivial whatsoever; however, I plan to undertake this work at short and medium term.

Another interesting research line to complement this work is to take into account the energy consumption of the reconfigurations during the scheduling process. The scheduler proposed in this doctoral thesis already tackles this problem, since it greatly reduces the number of required reconfigurations, even though it is not its primary objective. However, if the energy penalty was still very high, it would be interesting to modify the proposed scheduler in order to maximize task reuse, which has a direct impact on the energy consumption related to the reconfiguration process. In addition, increasing the task reuse involves other positive effects on the system performance. For instance, higher reuse rates also reduce the pressure over the external configuration memory and the system bus, since the reconfigurations involve moving large amounts of data from an external memory to the FPGA. In this sense, I am currently working on adapting the techniques proposed in this doctoral thesis in order to maximize task reuse in highly dynamic environments, where LFD cannot be directly applied. These optimizations involve changing the priority scheme of the replacement technique proposed in this doctoral thesis.

An additional issue that arises when working in the optimization of the energy consumption in reconfigurable systems is the secretiveness of some companies. In fact, depending on the manufacturer, in some cases is really difficult to obtain official and reliable figures about the energy consumption of a reconfigurable device when a reconfiguration process is being carried out. However, there are two possibilities to obtain this information. On the one hand, the reconfiguration circuitry can be modelled through a set of equations and then, its energy consumption can be estimated. On the other hand, an experimental measurement of the energy consumption can be directly obtained in a laboratory.

The current version of the scheduler developed in this doctoral thesis performs the replacements following a fixed priority scheme in order to achieve high performance. On the other hand, the also fixed “low power” priority scheme hinted above would configure the scheduler to minimize energy consumption. Hence, another possible extension of this work would be to make the system able to switch at run time between these two modes. This would give the scheduler the capability of offering two different trade-offs between performance and energy consumption, which is a very interesting feature.

Finally, we are also currently working on an extension of this work in order to deal with applications that cannot be represented using DAGs (for instance because they include pipelines). This is interesting since it will extend the applicability of the scheduling techniques presented in this doctoral thesis.

Le temps est un grand maître; le malheur est qu'il soit un maître inhumain qui tue ses élèves

Louis Hector Berlioz

Appendix A: Detailed description of a PLB peripheral created in EDK

This appendix describes in detail the inner structure of a slave PLB Peripheral created with the Xilinx™ EDK 9.1.02i design tool. It complements the description given in Subsection 5.1.2 about the peripheral that has been developed in EDK in order to integrate the proposed hardware scheduler in a microprocessor-based system for testing and evaluation purposes. As shown in Figure A.1, such peripheral is composed of a *PLB Intellectual Property Interface* (IPIF) and a *User Logic*, which are connected through an *Intellectual Property Interconnection* (IPIC)[17]. This appendix describes in detail the IPIF and the IPIC.

[17] *PLB IPIF (v2.02a)*: http://www.xilinx.com/support/documentation/ipembedprocess_coreconnect_plb-ipif.htm

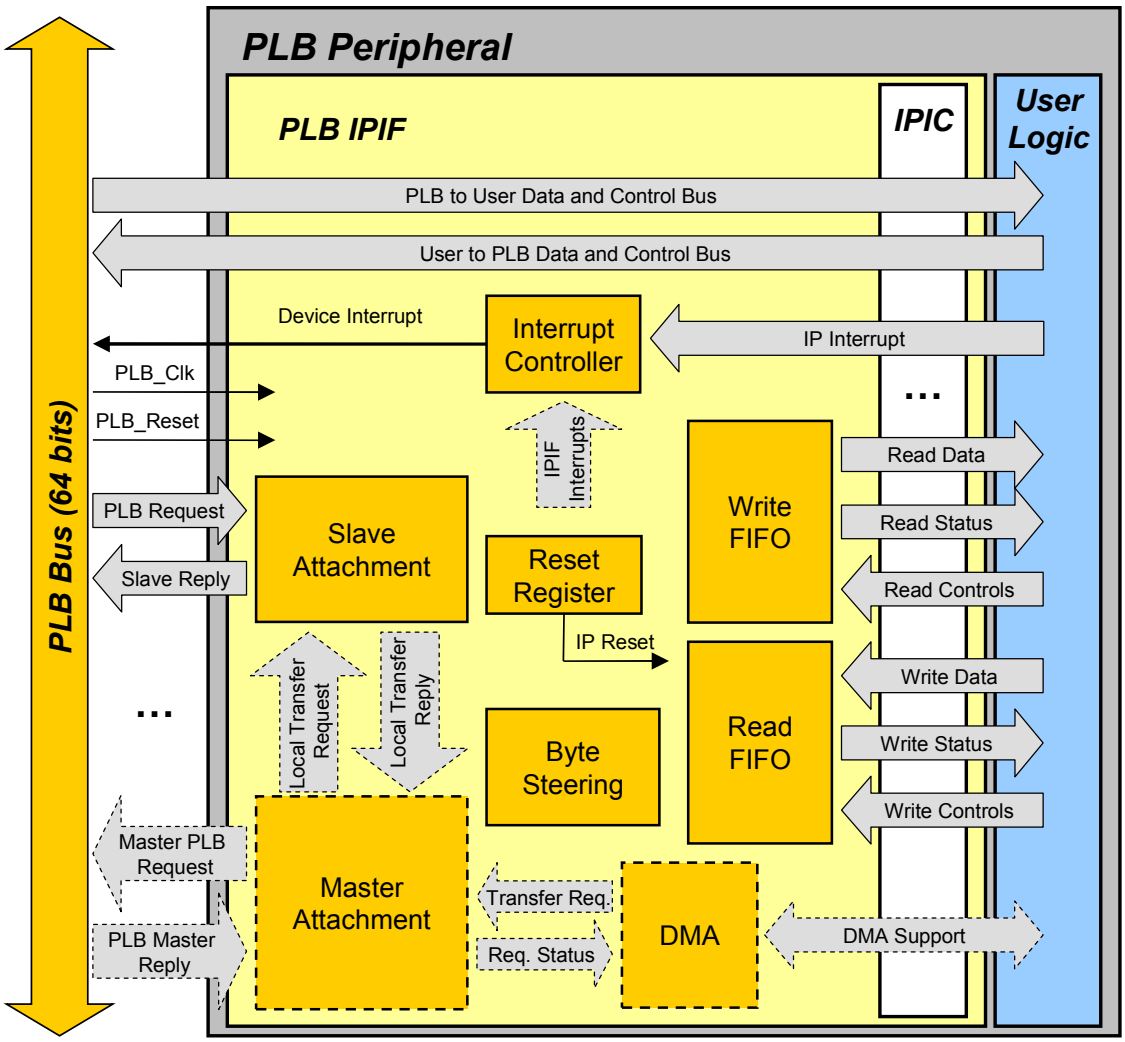

Figure A.1. Scheme of the developed PLB peripheral in EDK and detailed information of the PLB IPIF and the IPIC

The IPIC comprises the set of external signals to which the *User Logic* has access. As Figure A.1 shows, all of them are connected to any module existing inside the IPIF, except a pair of buses that are directly connected to the PLB bus. These are identified in the figure as: *PLB to User Data and Control Bus* and *User to PLB Data and Control Bus*. The former allows the user logic to receive data directly coming from the bus, whereas the latter does likewise in the opposite direction. Both buses are actually subsets of the communication lines that the

PLB includes, and the IPIF is built in such a way that the *User Logic* has direct access to them. There are more signals than those that are shown in Figure A.1 and that connect the PLB IPIF with the PLB bus and the *User Logic*. However, they do not appear in the figure for simplicity.

On the other hand, the IPIF includes a series of hardware blocks that implement the features that are described below. The base elements are the *Slave Attachment* and *Byte Steering* blocks, and are always present in every peripheral. The remaining ones can be optioned in or out. However, all of the shown elements in the figure are always present in this particular design, except the *DMA* and the *Master Attachment* modules, which are optional. This is indicated in Figure A.1 by drawing these modules and their corresponding input and output signals with dashed lines.

All the services that these modules implement are accessible by the Power PC by means of a set of software libraries that are included in EDK by default. A brief description of each one of these modules follows.

- The ***Slave Attachment*** module implements the protocol and timing translation between the PLB bus and the IPIC.
- The ***Byte Steering*** block is responsible for steering the data bus information onto the correct byte lanes. This block supports both read and write directions and is required when a target addresses space in the IPIF or the user IP has a data width smaller than the PLB bus width (64 bits).
- The ***Reset Register*** allows the system microprocessor to perform a local reset of the user IP; in this case, our scheduler.
- The inner ***Interrupt Controller*** collects the interrupts from the scheduler and the internal IPIF interrupt sources and stores it in the *Interrupt Storage Register* (Figure A.2). It then coalesces them to a single interrupt output signal that is directly connected to the microprocessor.

For this purpose, this module includes another register (the *Interrupt Enable Register*), which specifies whether an interrupt is enabled or not. This can be configured by the user. In this case, it is 1-bit wide since the scheduler only generates one type of interrupt. The contents of this register are used as a mask for the interrupt signals by means of an 'AND' gate, as shown in the figure. The output of this gate constitutes the output of the *Interrupt Controller*. In the case of the peripheral developed in this doctoral thesis, the *Interrupt Controller* receives one interrupt line from the *User Logic* and, if the peripheral implements the DMA service, another additional interrupt line.

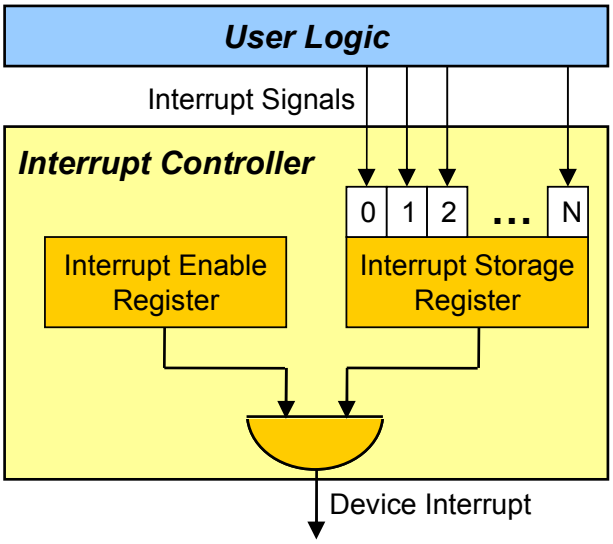

Figure A.2. Inner structure of the *Interrupt Controller* existing in the PLB IPIF

- The ***Read FIFO*** and the ***Write FIFO*** are two queues that implement the "First In First Out" data management policy and that are used to read and write data from/to the peripheral, respectively. They are synchronous and have user parameterized width and depth. In our case, the width has been set to 32 bits (the maximum in the 9.1.02i version of

EDK) and the depth, to 128 entries, which is more than enough to store all the data in the performed experiments.

- The PLB IPIF provides an optional ***Direct Memory Access*** (DMA) Service. This service automates the movement of large amounts of data between the *User Logic* or the IPIF FIFOs and other PLB peripherals. The inclusion of the DMA service automatically includes the *Master Attachment* module, since the DMA module requires access to the PLB as a master device. This module can be optionally included in the developed peripheral and it greatly speeds up the data transactions between the scheduler and the microprocessor, but at the cost of an additional hardware implementation cost. Chapter 6 includes comparative results of the performance and resources consumption of both implementation options.

All men make mistakes, but only fools persevere in the same error

Marcus Tullius Cicero

Apéndice B: Resumen en español

En cumplimiento del Artículo 4 de la normativa de la Universidad Complutense de Madrid que regula los estudios universitarios de posgrado, se presenta a continuación un resumen en español de la presente tesis que incluye la introducción, objetivos, principales aportaciones y conclusiones del trabajo realizado.

Este apéndice está organizado de la siguiente manera: La sección B.1 tiene como objetivo ofrecer una visión general del entorno sobre el que se basa la investigación realizada. De esta forma, se introducirá el hardware reconfigurable, para luego describir el entorno de trabajo que se ha utilizado durante la realización de la presente tesis doctoral, así como sus principales contribuciones. A continuación, la sección B.2 describe el algoritmo de planificación desarrollado. La implementación de este planificador se describe en

detalle en la sección B.3, mientras que los resultados experimentales se presentan en la sección B.4. Por último, la sección B.5 concluye este apéndice comentando posibles líneas de trabajo futuro.

B.1. Introducción

En los últimos años, estamos asistiendo a un crecimiento significativo en los requisitos computacionales de muchas aplicaciones. Entre éstas, destacan especialmente las aplicaciones multimedia; por ejemplo, JPEG, MPEG y MPEG [HW90, Noll97, Mark92]. Esto ha motivado el hecho de que, últimamente, muchos sistemas empotrados incluyan soporte específico para un gran número de aplicaciones, como codificadores y decodificadores de audio y vídeo, aplicaciones de renderización de gráficos en 3-D, aplicaciones de tratamiento de señal... Por ejemplo, en un computador personal estas aplicaciones se suelen optimizar utilizando tarjetas de vídeo y sonido, aceleradores gráficos y tarjetas de red inalámbricas, las cuales incluyen normalmente ASICs (en inglés "*Application-Specific Integrated Circuits*", o circuitos integrados para aplicaciones específicas) para alcanzar las prestaciones deseadas. Sin embargo, la falta de flexibilidad de estos circuitos específicos los hace inapropiados en muchos contextos, especialmente en sistemas empotrados, donde normalmente no hay suficiente espacio para incluir todos estos elementos.

Por este motivo, el hardware reconfigurable (HWR) en general y las FPGAs en particular han aparecido en las dos últimas décadas como una tecnología prometedora capaz de aunar en el mismo dispositivo la flexibilidad y la potencia de cálculo que muchas de estas aplicaciones requieren, tal y como se muestra en la figura B.1. Sin embargo, una de las mayores desventajas de usar hardware reconfigurable es que las latencias de reconfiguración de las tareas en los recursos hardware del dispositivo suelen ser muy significativas (del orden de cientos de milisegundos). Esto implica que, si se demandan muchas reconfiguraciones en relativamente poco tiempo, el rendimiento del sistema puede disminuir drásticamente. Es, por tanto, necesario solventar este problema si se quiere aprovechar al máximo las ventajas de este tipo de sistemas.

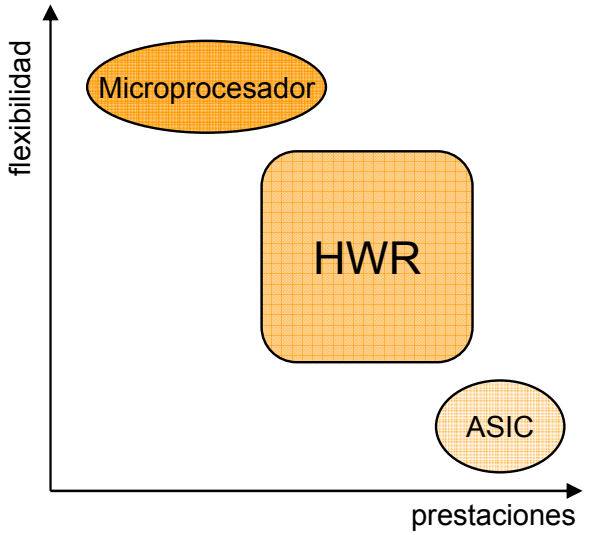

Figura B.1. Clasificación de las diferentes tecnologías en términos de flexibilidad y prestaciones

A nivel académico, se han propuesto muchas técnicas para reducir el impacto negativo que estas reconfiguraciones dinámicas tienen en el rendimiento del sistema. Una de las más interesantes y más estudiadas en la literatura consiste en tener en cuenta explícitamente el retardo de las reconfiguraciones durante el proceso de planificación de las aplicaciones en el dispositivo reconfigurable en cuestión.

Por otro lado, una de las principales características de muchas aplicaciones actuales es su alto nivel de dinamismo. En el contexto de esta tesis doctoral, eso significa que, en un instante de tiempo dado, la secuencia de reconfiguraciones que se van a demandar en el futuro se desconoce parcialmente o en su totalidad. Por tanto, en tiempo de diseño muchas veces es imposible saber cuál será la secuencia total de tareas que se ejecutarán en los recursos reconfigurables.

Este comportamiento tan impredecible hace que resulte muy difícil tomar buenas decisiones de planificación en tiempo diseño. En este caso, las dos únicas maneras de abordar tal nivel de dinamismo son:

- disponer de un sistema con los recursos necesarios para conseguir los requerimientos de rendimiento mínimos incluso en el caso peor.
- hacer predicciones sobre qué tareas se ejecutarán en el futuro, según la información que ha debido ser recopilada en ejecuciones previas.

Sin embargo, ambas soluciones tienen grandes desventajas:

- En sistemas empotrados normalmente es imposible incluir tal cantidad de recursos debido a su alto precio o consumo de energía.
- Las predicciones pueden ser imprecisas, lo cual conlleva a planificaciones subóptimas e incluso de mala calidad.

Por tanto, en este tipo de entornos dinámicos, la única alternativa que tenemos para conseguir obtener buenas planificaciones es planificar, al menos parcialmente, en tiempo de ejecución. Sin embargo, si el planificador realiza muchas operaciones en tiempo de ejecución, éstas pueden generar importantes penalizaciones en las aplicaciones que se están ejecutando. Por tanto, estas operaciones deben optimizarse en la medida de lo posible para evitar que el rendimiento del sistema se degrade aún más.

B.1.1. Objetivos y contribuciones de esta tesis

En el trabajo desarrollado en esta tesis doctoral propongo un planificador de tareas híbrido, parte del cual se ejecuta en tiempo de diseño y parte del cual se ejecuta en tiempo de ejecución. Este enfoque combina las mejores características de ambos esquemas por separado. El motivo es que por un lado, el hecho de tomar algunas decisiones en tiempo de ejecución hace que el

planificador sea capaz de adaptarse a eventos dinámicos e inesperados. Y por otro lado, el hecho de que la mayor parte de los cálculos se realicen en tiempo de diseño disminuye la carga computacional del sistema y evita degradaciones en el rendimiento de las aplicaciones. Con este planificador, pretendo dar soporte a un sistema multitarea basado en hardware reconfigurable para conseguir planificaciones de alta calidad, así como acelerar las cálculos que están involucradas en este proceso de planificación.

Las contribuciones presentadas en esta tesis doctoral son las siguientes:

La primera gran contribución de esta tesis es el **desarrollo de un algoritmo de planificación híbrido** que realiza la mayor parte de sus cálculos en tiempo de diseño, pero aun así mantiene suficiente flexibilidad para adaptarse a eventos inesperados en tiempo de ejecución. Las aplicaciones se ejecutan en un sistema multitarea compuesto por un conjunto de unidades reconfigurables de igual tamaño. Estas aplicaciones vienen representadas como DAGs (en inglés "*Directed Acyclic Graphs*" o grafos dirigidos acíclicos), cuyos nodos representan tareas computacionales y cuyas aristas representan dependencias de datos entre dos nodos. Asimismo, asumimos que cada nodo incluye una estimación de su tiempo de ejecución, que en cada instante sólo se puede asignar una tarea por unidad reconfigurable, y que todas las tareas que se ejecutan caben en las unidades reconfigurables del sistema. Los grafos de tareas implicados se ejecutan secuencialmente en el sistema.

Básicamente, el planificador propuesto se ejecuta en dos etapas. En primer lugar (etapa en tiempo de diseño), el grafo de tareas entrante se analiza para así obtener información útil sobre las tareas que lo componen. De este modo, para cada grafo de tareas, el planificador caracteriza todas sus tareas con tres parámetros: **peso**, **criticalidad** y **movilidad.** Seguidamente, en tiempo de ejecución se realiza la planificación propiamente dicha del grafo de tareas, utilizando la información que se obtuvo en tiempo de diseño. Para ello, el planificador en tiempo de ejecución carga las tareas con antelación, solapando así sus tiempos de reconfiguración con los tiempos de ejecución de otras tareas.

Esta técnica se conoce habitualmente como prebúsqueda [Hauc98, LH02]. Por defecto se sigue un esquema en el que las tareas se cargan tan pronto como sea posible. Sin embargo, en ciertas situaciones el planificador en tiempo de ejecución puede decidir retrasar algunas reconfiguraciones para evitar caer en óptimos locales, siempre que ello no genere un retardo adicional. Para ello se utiliza la **movilidad** de la tarea cuya reconfiguración se quiere retrasar.

Asimismo, he desarrollado un módulo de reemplazo y lo he integrado en el planificador en tiempo de ejecución para decidir qué tareas deben eliminarse del sistema para hacer sitio a las nuevas. Este módulo implementa una estrategia de reemplazo que analiza los candidatos para reemplazo disponibles y decide en tiempo de ejecución la víctima de reemplazo, intentando en la medida de lo posible no reemplazar candidatos críticos o candidatos que vayan a ser reutilizados próximamente. Para ello, el módulo de reemplazo utiliza la **criticalidad** de los candidatos disponibles.

Según los experimentos realizados, nuestro planificador oculta hasta un 99% de las latencias de reconfiguración iniciales de las tareas en experimentos consistentes en ejecuciones recurrentes de un conjunto de *benchmarks* extraídos de aplicaciones multimedia actuales. Además, la estrategia de reemplazo desarrollada obtiene mejores resultados en rendimiento respecto a otras estrategias de reemplazo existentes, tales como *Least Recently Used* (LRU) o *Longest Forward Distance* (LFD). Esta última estrategia de reemplazo obtiene la tasa de reutilizaciones óptima [Bela66]. Sin embargo, sólo puede aplicarse en contextos estáticos ya que necesita conocer la secuencia completa de grafos de tareas que se ejecután. Por otro lado, si el planificador retrasa algunas reconfiguraciones dependiendo de su movilidad, *las tasas de reutilización alcanzadas son incluso mejores que las que se consiguen con LFD*. El motivo por el que se consiguen estos excelentes resultados es el hecho de que LFD no puede retrasar ninguna reconfiguración; sin embargo, la estrategia de reemplazo propuesta aprovecha esta optimización.

La segunda gran contribución de la presente tesis doctoral es **una implementación eficiente del planificador en tiempo de ejecución**, utilizando una pequeña cantidad de los recursos reconfigurables existentes en la FPGA que se ha utilizado, una Virtex-II Pro XC2VP30. El objetivo es realizar los cálculos en tiempo de ejecución tan rápido como sea posible, para así generar la menor penalización en tiempo de ejecución posible (en este caso en particular, solamente unas decenas de ciclos de reloj). Para testear este módulo hardware, lo he integrado en una plataforma de simulación desarrollada e implementada también para la FPGA. Esta plataforma utiliza un conjunto de temporizadores programables para simular el comportamiento de las unidades reconfigurables. Dicho sistema también es capaz de medir cuántos ciclos de reloj han sido necesarios para la ejecución de las operaciones del planificador en tiempo ejecución.

Finalmente, en esta tesis doctoral también presento una comparación de la implementación hardware del planificador en tiempo de ejecución con una versión software equivalente. Esta última consiste en un programa escrito originalmente en el lenguaje de programación C y compilado para uno de los microprocesadores empotrados Power PC que existen en la Virtex-II Pro utilizada. Los resultados de esta comparativa muestran que la versión hardware es hasta tres órdenes de magnitud más eficiente que la versión software equivalente. Sin embargo, la gran ventaja de esta última es el hecho de no utilizar recursos reconfigurables adicionales. Por tanto, estas dos versiones ofrecen diferentes compromisos entre las latencias que se generan en tiempo de ejecución y el coste para implementar el circuito hardware.

B.1.2. Arquitectura objetivo

La figura B.2 muestra la arquitectura objetivo para la que se ha diseñado el planificador propuesto en esta tesis. La región etiquetada como "*HW reconf.*" incluye tanto la implementación hardware de nuestro planificador de tareas,

como el conjunto de unidades reconfigurables del sistema (*URs*), de igual tamaño.

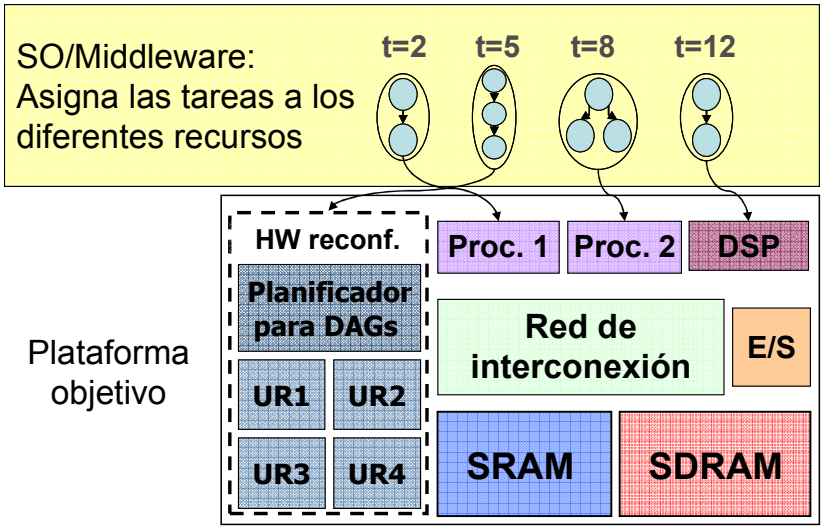

Figura B.2. Arquitectura objetivo

Las tareas se cargan en las unidades reconfigurables utilizando reconfiguración parcial en tiempo de ejecución. Para proporcionar el soporte necesario para que las tareas asignadas a estas unidades se comuniquen con el resto del sistema se ha utilizado el modelo ICN (*InterConnection Network*) [MBV+02, MMB+03] en el que cada unidad reconfigurable incluye un interfaz de comunicación fijo que permite ejecutar las primitivas de comunicación básicas de paso de mensajes. Asimismo, estos interfaces proporcionan la funcionalidad básica que necesitan el planificador de tareas y el sistema operativo para dirigir la ejecución de las tareas en estas unidades.

Todas las unidades reconfigurables se comunican entre ellas a través de una infraestructura de comunicación fija (etiquetado en la figura B.2 como "*Red de interconexión*"). En el desarrollo de esta tesis, se ha asumido que dicha infraestructura se ha diseñado a prueba de conflictos y proporciona el ancho de banda suficiente de modo que incluso las comunicaciones en el caso peor se

realizan de forma correcta y eficiente. Esto se puede conseguir, por ejemplo, utilizando un bus que soporte el máximo ancho de banda requerido, o utilizando una *Network-on-Chip* (NoC) [DMB02] diseñada a prueba de conflictos. Para ello, dicha NoC podría implementar un algoritmo de rutado basado en *wormhole* o una técnica similar que garantice que la latencia de un mensaje transmitido entre cualesquiera dos nodos sea prácticamente constante, con independencia de la distancia entre ellos. Por tanto, bajo estas suposiciones, podemos asumir con total seguridad que cualquier tarea puede ejecutarse correctamente en cualquiera de las unidades reconfigurables disponibles.

La región reconfigurable de nuestro sistema solamente incluye un circuito de reconfiguración, de forma similar a todas las FPGAs comerciales que existen en el mercado. Esto significa que este circuito solamente puede realizar una reconfiguración a la vez, no varias en paralelo. Además, en este sistema se puede realizar reconfiguración parcial de tal manera que este proceso no interfiere en absoluto en la ejecución del resto de las tareas que ya se están ejecutando en el sistema. El planificador de tareas desarrollado tiene en cuenta explícitamente estas dos características del sistema para conseguir buenos resultados de planificación. Para ello, lanza las reconfiguraciones y ejecuciones de las tareas en paralelo siempre que sea posible.

Tal y como se muestra en la figura B.2, la arquitectura objetivo también incluye otros elementos de proceso (procesadores y DSPs), bloques de memoria SRAM y SDRAM que constituyen la jerarquía de memoria y controladores de entrada/salida (*E/S*). Finalmente, controlando todos estos elementos, existe un sistema operativo o un *middleware* que proporciona la información necesaria acerca de las aplicaciones que se ejecutarán y que, dependiendo de sus restricciones temporales, decide si una aplicación debe ser asignada a un procesador empotrado, a un DSP o a una región reconfigurable. Y en este último caso, el planificador de tareas intentará ejecutar estas tareas de la forma más eficiente posible, minimizando sus latencias de reconfiguración.

B.2. El algoritmo de planificación propuesto

Como se mencionó anteriormente, el algoritmo de planificación propuesto está dividido en dos etapas, las cuales se ejecutan en tiempo de diseño y en tiempo de ejecución, respectivamente. Las dos siguientes subsecciones describen en detalle cada una de ellas.

B.2.1. Planificador en tiempo de diseño

En tiempo de diseño, el planificador obtiene información útil de las tareas que componen el grafo de tareas que se ejecutará. Esta información está compuesta de tres parámetros: pesos, criticalidad y movilidad. Cada uno de estos parámetros se calcula de forma separada, tal y como se muestra en la figura B.3.

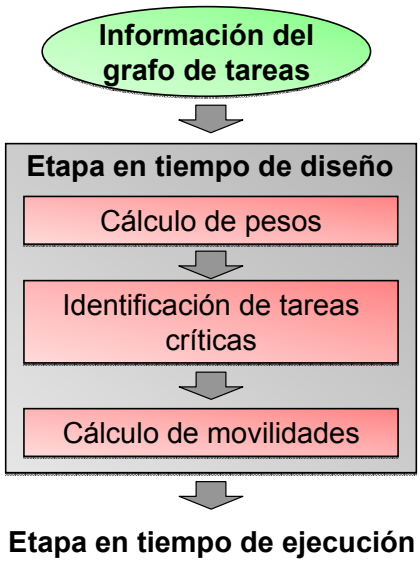

Figura B.3. Organigrama de la etapa en tiempo de diseño del planificador propuesto

En primer lugar se obtiene el **peso** de las tareas, que se utiliza para obtener la secuencia de reconfiguraciones asociadas al grafo de tareas en cuestión. La idea es reconfigurar en primer lugar aquellas tareas que tengan un mayor impacto en el camino crítico del grafo. Para ello, se sigue la fórmula descrita en (1):

$$(1) \qquad P(t_i) = tiempo\_ejecucion(t_i) + MAX\{P(t_j)\}, \forall t_i \in TAREAS, \forall t_j \in SUCESORES(t_i)$$

Esta fórmula significa que el peso de un nodo hoja (es decir, sin sucesores) es simplemente su tiempo de ejecución; y para el resto de las tareas, su peso es la suma de su tiempo de ejecución y el máximo de los pesos de todos sus sucesores. Una vez que se realiza este cálculo, las tareas se ordenan de forma decreciente según sus pesos para obtener la secuencia de reconfiguraciones que el planificador seguirá en tiempo de ejecución.

En segundo lugar, para cada tarea, se obtiene su **criticalidad**, que evalúa la latencia adicional que se genera cuando esa tarea no se reutiliza. En concreto, el objetivo de esta etapa es obtener el *mínimo* conjunto de tareas que cumplen la siguiente condición:

Si esas tareas se reutilizan (y por tanto, no generan ningún retardo adicional debido a sus latencias de reconfiguración), el planificador es capaz de ocultar las latencias de reconfiguración de las tareas restantes (es decir, no críticas).

Una vez que el conjunto de tareas críticas se ha obtenido, a cada una de ellas se le asigna un valor de criticalidad, que es precisamente el retardo adicional que se genera durante la ejecución del grafo de tareas cuando esa tarea no se reutiliza. Por tanto, la criticalidad de las tareas no críticas es 0.

Finalmente, la **movilidad** se utiliza para retrasar la carga de algunas reconfiguraciones con respecto a un esquema en el que las tareas se cargan tan pronto como sea posible, teniendo en cuenta la naturaleza dinámica del sistema

y las dependencias del grafo actualmente en ejecución. El objetivo es evitar que el planificador devuelva soluciones subóptimas. Este parámetro viene dado en términos de cuántos eventos el planificador en tiempo de ejecución puede saltar antes de reconfigurar sin que se genere un retardo adicional, con respecto a un enfoque en el que las cargas y las ejecuciones de las tareas se realizan ASAP (en inglés "*As Soon as Possible*", tan pronto como sea posible).

La figura B.4 muestra un ejemplo que motiva esta idea. Esta figura muestra la ejecución que se obtiene cuando se aplica un enfoque puramente ASAP (b) y cuando el planificador retrasa algunas reconfiguraciones (c), para el ejemplo de la figura B.4.a. En este ejemplo, vemos que para la ejecución ASAP, la tarea 4 reemplaza a la tarea 1 inmediatamente después de la ejecución de esta última, ya que el planificador intenta cargarla lo antes posible, y en ese instante de tiempo sólo está disponible la unidad reconfigurable 1 (donde la tarea 1 está cargada). Por este motivo, cuando el mismo grafo de tareas se ejecuta de nuevo, la tarea 1 tiene que volver a ser cargada, lo cual genera un retardo adicional de 4 milisegundos en su ejecución.

Sin embargo, esta penalización desaparece si se retrasa la reconfiguración de la tarea 4 (Figura B.4.c). En este caso, cuando la tarea 1 es seleccionada como víctima de reemplazo (tiempo = 16 milisegundos), el planificador decide retrasar esta reconfiguración y espera al final de la ejecución de la tarea 2. De este modo, cuando la tarea 2 termina su ejecución, el módulo de reemplazo puede seleccionar entre dos posibles candidatos (las tareas 1 y 2, en las unidades reconfigurables 1 y 3, respectivamente), y seleccionará la UR3 porque la tarea 2 no es crítica. Por tanto, cuando el mismo grafo de tareas se ejecuta otra vez, la tarea 1 se reutiliza y en este caso ninguna reconfiguración genera retardos adicionales en el sistema.

Por otro lado, en este punto es importante destacar que en tiempo de ejecución, el planificador sigue un esquema basado en eventos. Es decir, las decisiones en tiempo de ejecución se toman cuando ocurren ciertos eventos en tiempo de ejecución, que se generan cuando viene un nuevo grafo de tareas, y

cuando se termina de reconfigurar o ejecutar una tarea (la siguiente subsección dará más detalles acerca de cómo funciona la gestión de estos eventos). En cualquier caso, la movilidad de una tarea se expresa en términos de cuántos eventos se puede retrasar su reconfiguración sin que esto genere una retardo adicional.

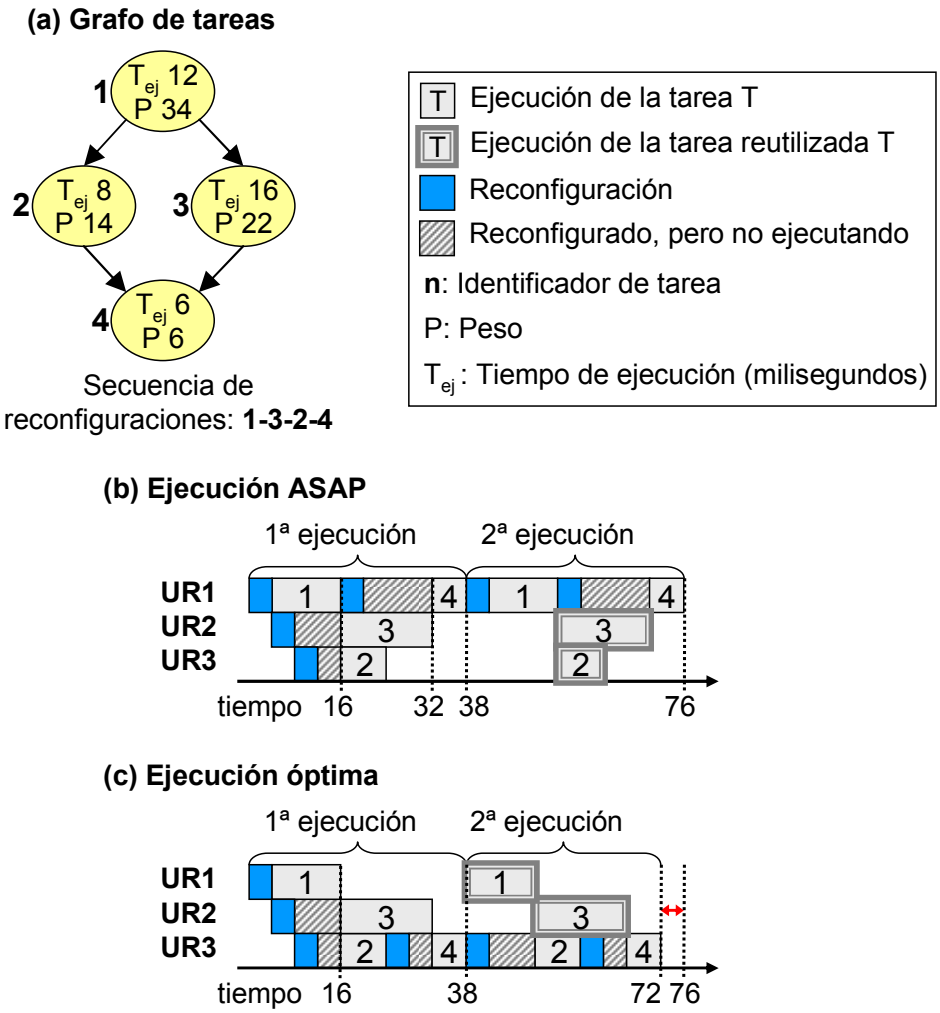

Figura B.4. Ejecución de un grafo de tareas (a) en una plataforma con tres URs aplicando una estrategia de planificación ASAP (b) utilizando nuestra técnica de planificación, la cual obtiene la solución óptima retrasando una de las reconfiguraciones (c). La latencia de reconfiguración es de 4 milisegundos

El motivo por el que el planificador decide retrasar una reconfiguración siempre es el mismo: la técnica de reemplazo ha seleccionado una víctima que el planificador decide no reemplazar. En este caso, el planificador comprueba la movilidad de la tarea implicada, y si aún hay margen para retrasar esa reconfiguración, esperará hasta el próximo evento con la esperanza de que en ese momento la técnica de reemplazo encontrará una víctima menos importante.

B.2.2. Planificador en tiempo de ejecución

El planificador en tiempo de ejecución guía la ejecución de los grafos de tareas teniendo en cuenta las dependencias internas de sus tareas, la información que obtuvo el planificador en tiempo de diseño y el estado dinámico del sistema. Para ello, este planificador explota la prebúsqueda y la reutilización de las tareas para ocultar las latencias de reconfiguración durante la ejecución de los grafos de tareas.

Tal y como se muestra en la figura B.5, el planificador en tiempo de ejecución está compuesto por dos módulos que interactúan entre ellos y realizan la *gestión de la ejecución del grafo de tareas* y la *técnica de reemplazo*, respectivamente.

El módulo que realiza la gestión de la ejecución del grafo de tareas garantiza la correcta ejecución de estos grafos teniendo en cuenta las dependencias de datos existentes entre sus tareas y los recursos disponibles. Para ello, y como ya se mencionó anteriormente, este módulo sigue una estrategia basada en la generación de eventos en ciertos instantes significativos durante la ejecución del grafo de tareas. Este enfoque reduce en gran medida la complejidad de la planificación en tiempo de ejecución. Los eventos que se generan y capturan son los siguientes:

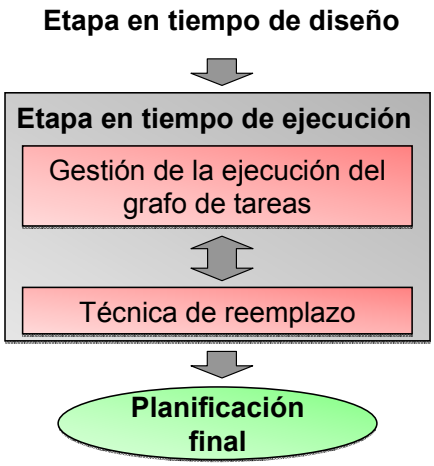

Figura B.5. Organigrama de la etapa en tiempo de ejecución del planificador propuesto

- ***nuevo_grafo_de_tareas*:** este evento se genera cuando un nuevo grafo de tareas tiene que ser ejecutado.
- ***fin_de_ejecucion*:** este evento se genera cada vez que se termina de ejecutar una tarea.
- ***fin_de_reconfiguracion*:** este evento se genera cada vez que se termina de reconfigurar una tarea.
- ***tarea_reutilizada*:** este evento se genera cuando se reutiliza una tarea. Esto ocurre cuando el planificador intenta cargar una tarea e identifica que ya estaba cargada en el sistema.

Los eventos *fin_de_reconfiguracion* y *tarea_reutilizada* son similares desde el punto de vista del planificador, ya que reutilizar una tarea es equivalente a llevar a cabo una reconfiguración en cero ciclos de reloj. Estos eventos se generan en tiempo de ejecución se capturan y procesan secuencialmente. Dependiendo del tipo de evento que se procesa, se toman las decisiones que están descritas en la figura B.6.

Cuando un nuevo grafo de tareas llega para su ejecución (Línea 1) y el circuito de reconfiguración está libre (Línea 2), el sistema intenta planificar la siguiente tarea de la secuencia de reconfiguraciones (Línea 3).

```
/* tarea = tarea que disparó el evento
 * CR = Circuito de Reconfiguración */
CASE evento IS:
 1. nuevo_grafo_de_tareas:
 2.     IF (CR == libre){
 3.         buscar_reconfiguracion (&secuencia_rec);
 4.     }
 5. fin_de_reconfiguracion o tarea_reutilizada:
 6.     comprobar_dependencias (&tarea);
 7.     IF (preparada_para_ejecutar (&tarea)){
 8.         comienza_ejecucion (tarea);
 9.     }
10.    buscar_reconfiguracion (&secuencia_rec);
11. fin_de_ejecucion:
12.     IF (CR == libre){
13.         buscar_reconfiguracion (&secuencia_rec);
14.     }
15.     actualiza_dependencias (&tarea)
16.     FOR (i := 0 TO NUMERO_DE_URS){
17.         comprueba_dependencias (&tarea);
18.         IF (preparada_para_ejecutar (&tarea)){
19.             comienza_ejecucion (tarea);
20.         }
21.     }
```

Figura B.6. Pseudo-código del gestor de ejecución de los grafos de tareas

Para los eventos *fin_de_reconfiguracion* y *tarea_reutilizada* (Línea 5), el sistema comprueba las dependencias de la tarea que se acaba de cargar o reutilizar. Para ello, se comprueba si todos sus predecesores han terminado su ejecución (Líneas 6 y 7). Si es así, el planificador lanza su ejecución (Línea 8). A continuación, se intenta planificar la carga de la siguiente tarea en la secuencia de reconfiguraciones (Línea 10).

Por último, para el evento *fin_de_ejecucion* (Línea 11), el planificador comprueba de nuevo si el circuito de reconfiguración está libre. Si es así, intenta planificar la carga de la siguiente tarea (Líneas 12-13). A continuación, actualiza las dependencias del grafo, disminuyendo en uno el número de predecesores de la tarea cuya ejecución acaba de finalizar (Línea 15). Por último, el planificador comprueba si alguna de las tareas que están cargadas en ese momento en las unidades reconfigurables está preparada para su ejecución (Líneas 16-18). Si es así, se lanza su ejecución (Línea 19).

Por otro lado, en la función *buscar_reconfiguracion (&secuencia_rec)* (Líneas 3, 10 y 13), el planificador extrae la siguiente reconfiguración de la secuencia de reconfiguraciones y decide si cargarla o no. Para ello, invoca al módulo de reemplazo y si la víctima de reemplazo seleccionada es crítica y su movilidad es mayor que el número de eventos que se han saltado hasta ese momento, se retrasa la carga de esa tarea, al menos, hasta el siguiente evento. Este código no aparece en la figura B.6 por simplicidad. Esta técnica (a la que hemos llamado "*Skip Events*") evita enormemente que el planificador devuelva resultados subóptimos. Por lo tanto, las planificaciones obtenidas mejoran tanto a nivel de prestaciones como a nivel de número de tareas reutilizadas.

Finalmente, el módulo de reemplazo implementa la estrategia LF+C que se ha desarrollado específicamente para este planificador. Básicamente, esta estrategia intenta evitar reemplazar candidatos que sean críticos y/o que vayan a ser reutilizados en el futuro cercano (es decir, en el contexto del grafo de tareas actualmente en ejecución). Para ello, en función de esta información, los candidatos se clasifican en 6 categorías:

- **Candidatos ocupados:** Unidades reconfigurables con una tarea en ejecución, o con una reconfiguración que acaba de ser precargada y que está esperando para su ejecución. Estos candidatos nunca se seleccionan para ser reemplazados.

- **Candidatos libres (CLs):** Estas unidades reconfigurables no tienen ninguna tarea cargada como resultado de una ejecución previa. Todas las unidades reconfigurables son candidatos libres al inicio de la ejecución de un grafo de tareas.
- **Candidatos no críticos (CNCs):** Unidades reconfigurables con tareas no críticas y que no van a ejecutarse de nuevo en el futuro próximo.
- **Candidatos no críticos reutilizables (CNCRs):** Unidades reconfigurables con tareas no críticas y que van a ejecutarse de nuevo en el futuro próximo.
- **Candidatos críticos (CCs):** Unidades reconfigurables con tareas críticas que no van a ejecutarse de nuevo en el futuro próximo.
- **Candidatos críticos reutilizables (CCRs):** Unidades reconfigurables con tareas críticas que sí van a ejecutarse de nuevo en el futuro próximo.

La estrategia de reemplazo asigna la máxima prioridad a los CCRs y la mínima prioridad a los CLs. De esta manera, los primeros candidatos que LF+C intenta reemplazar son los CLs; y a continuación, CNCs, CNCRs, CCs y CCRs, en este orden. Si los mejores candidatos que LF+C encuentra son todos del mismo tipo, entonces se selecciona el primer candidato de ese tipo que se haya encontrado.

B.3. Detalles de implementación

Durante la realización de esta tesis doctoral, se ha desarrollado una implementación del planificador en tiempo de ejecución utilizando una pequeña cantidad de recursos reconfigurables de una FPGA de pequeñas dimensiones, una Virtex-II Pro XC2VP30. El objetivo de esta implementación es demostrar que se puede conseguir acelerar el proceso de planificación en tiempo de ejecución, generando así retardos despreciables que apenas introducen penalizaciones en la planificación y ejecución de las tareas.

Asimismo, se ha desarrollado una versión software equivalente del planificador en tiempo de ejecución, consistente en un programa escrito en C y compilado para ser ejecutado en uno de los Power PC empotrados en la Virtex-II Pro utilizada. Ambas versiones ofrecen diferentes compromisos entre prestaciones y consumo de recursos hardware, tal y como se mostrará en los resultados experimentales.

B.3.1. Implementación hardware

El planificador hardware desarrollado tiene la estructura descrita en la figura B.7. Contiene los siguientes módulos:

La *Tabla de Tareas* almacena la información acerca del grafo de tareas actualmente en ejecución. Sobre ella se pueden realizar operaciones de inserción, consulta y borrado de tareas, que el planificador realizará a medida que avance la ejecución del grafo de tareas. La operación de borrado también actualiza las dependencias internas del grafo.

Las *FIFOs de Reconfiguraciones y Eventos* guardan la secuencia de reconfiguraciones que se debe seguir y los eventos que se generan en tiempo de ejecución, respectivamente. Ambos módulos guardan y actualizan su

correspondiente información siguiendo una política de actualización FIFO (en inglés *First-In, First-Out*).

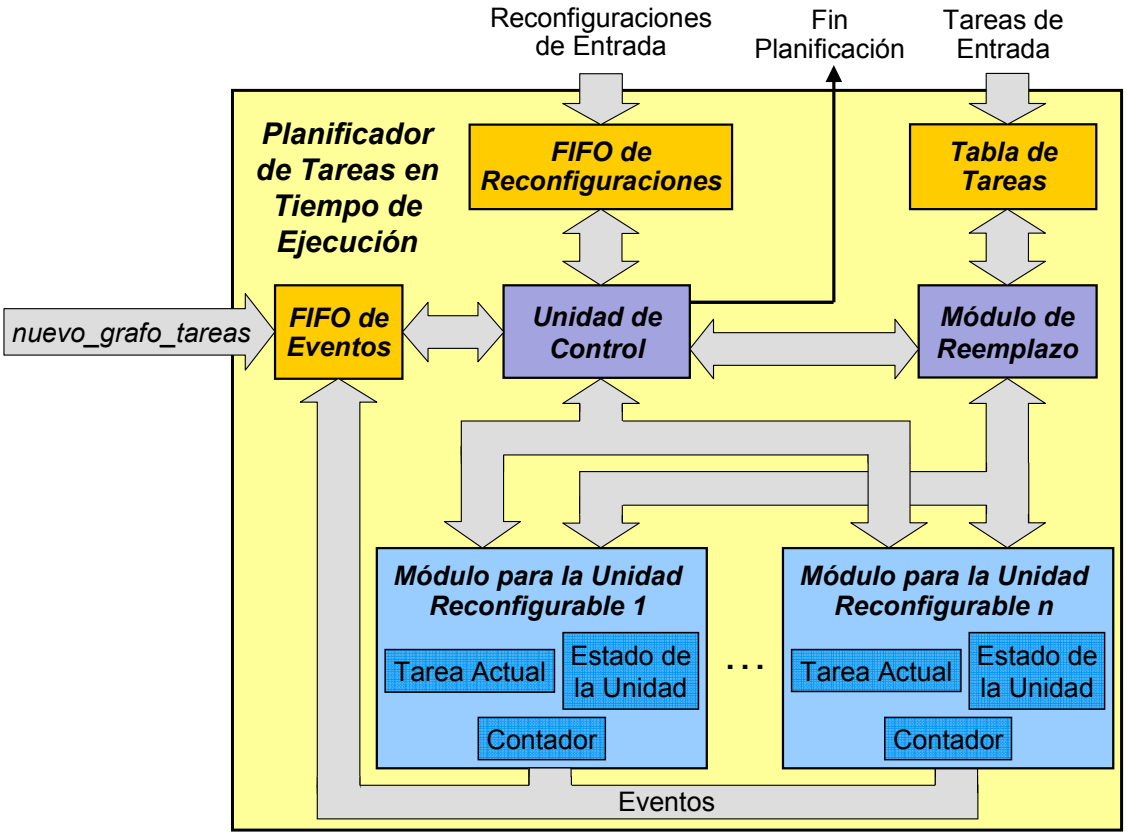

Figura B.7. Esquema de la implementación hardware propuesta de nuestro planificador en tiempo de ejecución

Los *Módulos para las Unidades Reconfigurables* simulan los tiempos de reconfiguración y ejecución a través de contadores, tal y como se muestra en la figura B.7. Además, cada módulo guarda el estado de la unidad, así como la tarea que está actualmente cargada en esa unidad reconfigurable. Estos módulos generan eventos en tiempo de ejecución cada vez que una tarea termina su reconfiguración o su ejecución.

El *Módulo de Reemplazo* decide en tiempo de ejecución en qué unidad cargar las tareas del grafo de tareas, teniendo en cuenta su criticalidad y si pueden ser reutilizadas en alguna unidad reconfigurable o no. Este módulo

implementa la estrategia de reemplazo LF+C desarrollada en esta tesis doctoral, la cual fue descrita en detalle en el apartado anterior.

Finalmente, la *Unidad de Control* gestiona el funcionamiento general del sistema. Básicamente, este módulo extrae secuencialmente los eventos que se han almacenado en la FIFO de eventos e interactúa con el resto de los módulos del sistema para tomar las decisiones de planificación adecuadas descritas anteriormente en la figura B.6.

Para poder evaluar sus prestaciones, este planificador se ha integrado en un sistema basado en microprocesador. Dicho sistema se ha desarrollado e implementado también para la Virtex-II Pro XC2VP30. Para ello, se ha utilizado la herramienta de desarrollo Embedded Development Kit (EDK) de Xilinx™. El sistema desarrollado aparece descrito en la figura B.8.

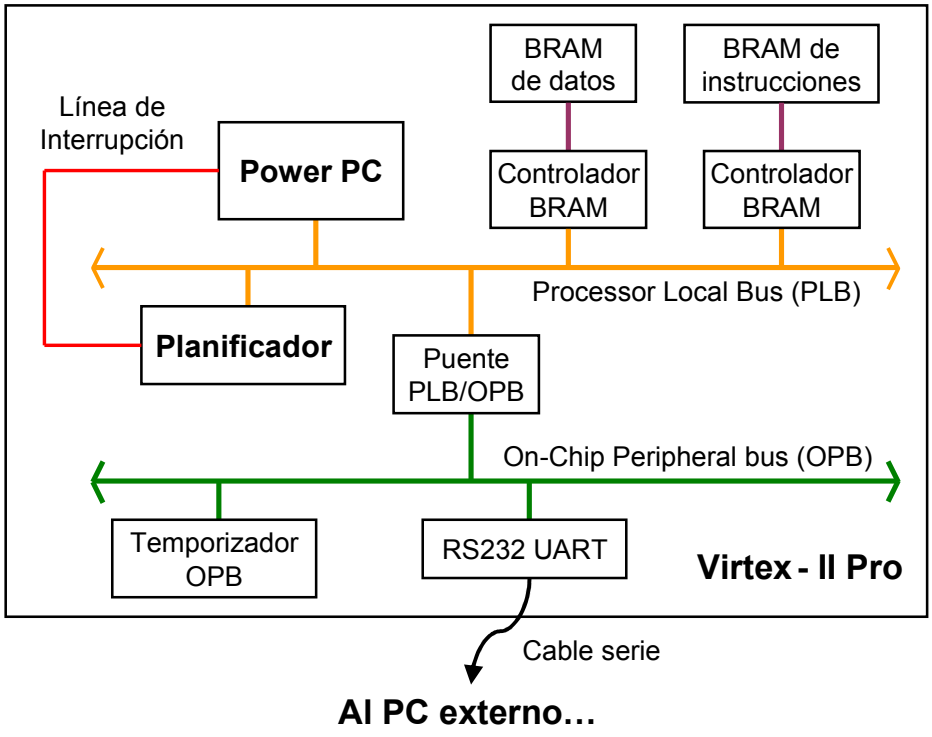

Figura B.8. Sistema basado en microprocesador en el cual se ha integrado el planificador hardware

Este sistema está compuesto de un procesador, una jerarquía de memoria con dos memorias para datos e instrucciones respectivamente, y un conjunto de periféricos, entre los cuales se encuentra nuestro planificador. Todos estos módulos están conectados o bien a un bus OPB (*On-Chip Peripheral Bus*) o a un bus PLB (*Processor Local Bus*). Ambos buses están conectados a través de un Puente PLB/OPB. El procesador tiene acceso directo al planificador a través del bus PLB, y ambos módulos están conectados también por medio de una línea de interrupción. De esta manera, el procesador le envía al planificador la información de los grafos de tareas que se ejecutarán, y el planificador avisa de su terminación activando la línea de interrupción.

Por último, el *Temporizador OPB* se utiliza para medir los retardos que el planificador ha generado durante la ejecución de los grafos de tareas, mientras que la *RS232 UART* permite conectar la placa a un PC externo a través de un puerto serie. Esto facilita la depuración y la obtención de resultados.

B.3.2. Implementación software

Como ya he mencionado anteriormente, durante el transcurso de esta tesis doctoral he desarrollado además una versión equivalente del planificador hardware que he descrito en la subsección anterior. Consiste en un programa escrito originalmente en el lenguaje de programación C y compilado para uno de los microprocesadores empotrados Power PC que existen en la Virtex-II Pro utilizada. Para evaluar su funcionalidad, dicho programa se ejecuta en el sistema basado en microprocesador que se muestra en la figura B.9.

De forma similar al sistema que fue descrito en la figura B.8, el sistema de la figura B.9 contiene un Power PC, una jerarquía de memoria y un conjunto de periféricos, todos ellos conectados al bus OPB. Asimismo, el *Temporizador OPB* se utiliza para evaluar las prestaciones de este planificador y la *RS232 UART*

conecta este sistema a un PC externo por puerto serie para depurar y obtener resultados.

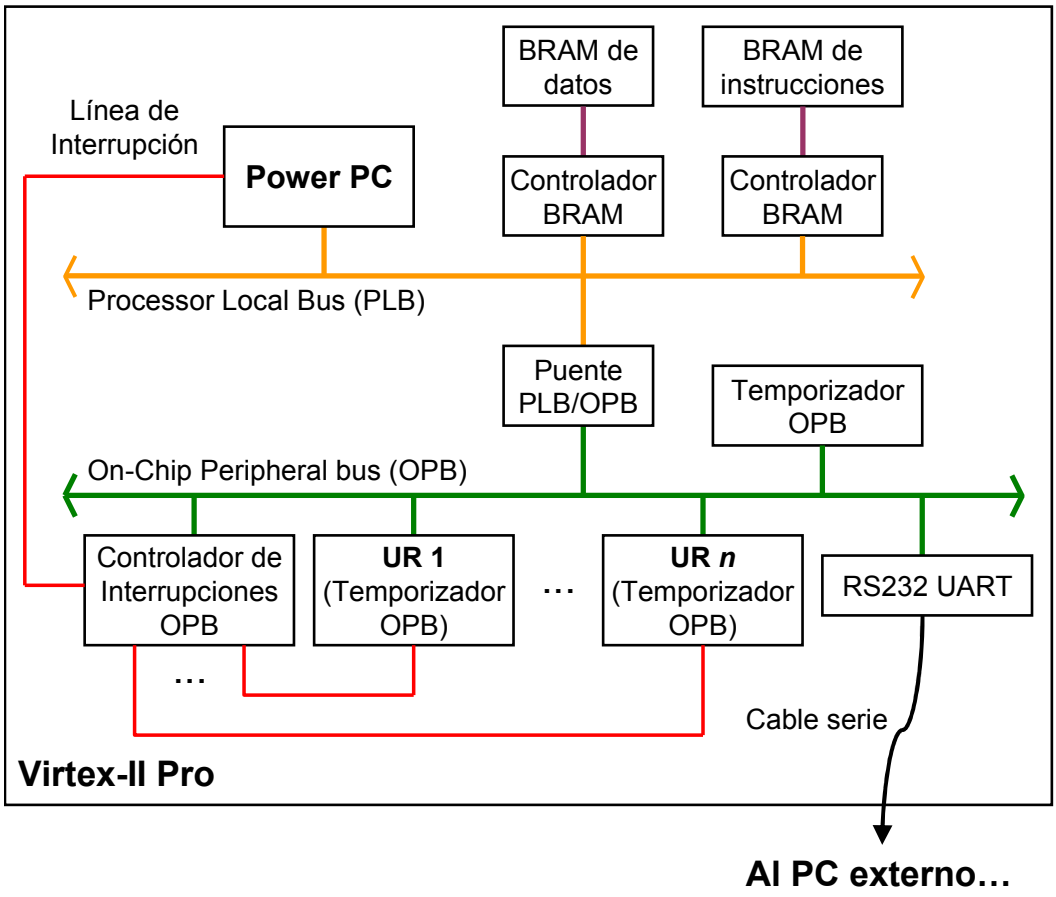

Figura B.9. Sistema basado en microprocesador en el cual se ha integrado el planificador software

Sin embargo, a diferencia del sistema de la figura B.8, en este caso no existe ningún periférico hardware que implemente la funcionalidad del planificador, ya que éste se ejecuta en el Power PC. Las latencias de reconfiguración y ejecución de las tareas en las unidades reconfigurables se simulan en este caso a través de sendos *Temporizadores OPB* adicionales (*UR 1, …, UR n*). De este modo, cada vez que hay que simular una latencia, el temporizador correspondiente inicia la cuenta de un determinado número de ciclos de reloj, y

cuando esta cuenta finaliza, se genera una interrupción. Dependiendo de si la latencia simulada es de reconfiguración o de ejecución, el planificador asociará esta interrupción a la generación de un evento *fin_de_reconfiguracion*, *fin_de_ejecucion*, o *tarea_reutilizada*. Y dependiendo del evento generado, el planificador software le dará su tratamiento adecuado, el cual fue descrito en la sección anterior (figura B.6). Los *Temporizadores OPB* que simulan las unidades reconfigurables están conectados al procesador a través de un *Controlador de Interrupciones*, cuya salida se conecta al Power PC. Este esquema permite al Power PC recibir y procesar múltiples interrupciones, ya que este procesador sólo tiene dos líneas de interrupciones de entrada.

B.4. Resultados experimentales

Los resultados obtenidos muestran que el planificador propuesto ayuda a obtener de forma eficiente planificaciones de alta calidad en entornos altamente dinámicos en los que la secuencia de tareas que se ejecutarán en el futuro es desconocida a priori. Además, las dos implementaciones hardware y software propuestas ofrecen diferentes compromisos entre consumo de recursos hardware y prestaciones. En las siguientes subsecciones se resumirán los principales resultados obtenidos durante el desarrollo de esta tesis doctoral: resultados de síntesis, un análisis de las penalizaciones que el planificador en tiempo de ejecución introduce y por último, una evaluación de las prestaciones de la técnica de planificación propuesta.

B.4.1. Resultados de síntesis

La figura B.10 muestra el consumo de recursos del planificador hardware para una Virtex-II Pro XC2VP30, y para diferentes números de unidades reconfigurables. La figura muestra que el consumo de recursos aumenta con el número de unidades reconfigurables; sin embargo, el consumo de recursos global es muy reducido, ya que la utilización de ninguno de los posibles tipos de recursos supera el 9% de los disponibles en la FPGA. En la figura sólo se muestra qué ocurre para un número de unidades reconfigurables que oscila entre 3 y 8. Sin embargo, el planificador es totalmente escalable y el número de unidades podría aumentar sin ningún problema siempre que hubiese recursos disponibles para su implementación.

La frecuencia de funcionamiento de este planificador es aproximadamente 103 MHz y apenas cambia cuando el número de unidades reconfigurables aumenta. Por tanto, ha sido posible integrarlo en un sistema basado en

microprocesador en el que tanto el procesador como los periféricos y los buses funcionan a 100 MHz.

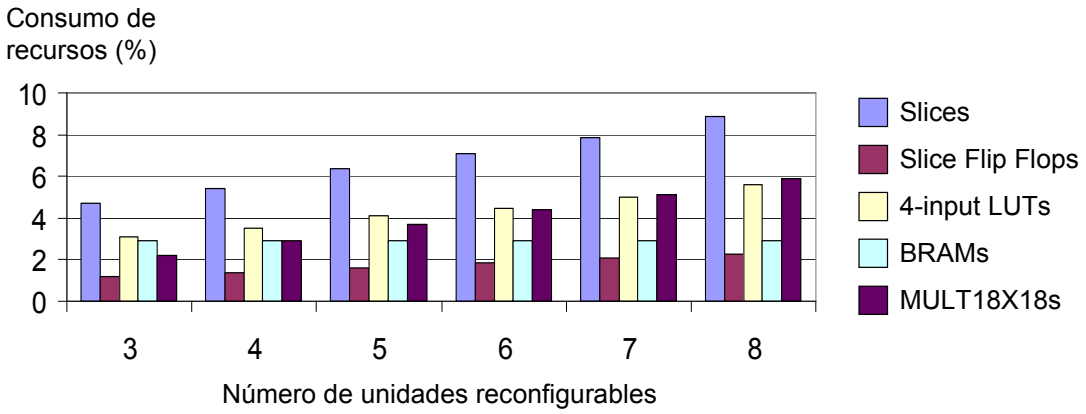

Figura B.10. Coste de implementación para el planificador hardware con diferentes números de unidades reconfigurables

Obviamente, la implementación software no tiene ningún consumo adicional de recursos hardware. Sin embargo, el tamaño del código que debe ejecutarse en el Power PC es mayor. Menciono esto porque en sistemas empotrados a veces es muy importante tener en cuenta este hecho, ya que normalmente la cantidad de memoria disponible suele ser muy reducida (por ejemplo, en una FPGA Virtex-II Pro XC2VP30 hay un total de tan sólo 359.5 KB de memoria RAM). En cualquier caso, el tamaño del fichero *.elf* correspondiente a la versión software es 118 KB, mientras que para la versión hardware este tamaño es 85.5 KB (pese a que el planificador no se ejecuta en el procesador, sigue haciendo falta un código de inicialización para los periféricos del sistema y para enviar los datos al planificador). En ningún caso esto ha representado problema alguno en la implementación de ambas versiones en la FPGA.

B.4.2. Penalizaciones en tiempo de ejecución introducidas por el planificador

La tabla B.1 muestra las penalizaciones que ambas versiones del planificador (hardware y software) introducen en la ejecución de diferentes aplicaciones. Para evaluar estas penalizaciones, se han seleccionado un conjunto de grafos de tareas extraídos de aplicaciones multimedia reales: dos versiones del decodificador JPEG (JPEG y Parallel-JPEG), el codificador MPEG-1, la transformada de Hough y la librería Pocket-GL [RMC05, RMVC05]. Esta última incluye 20 grafos de tareas que han sido agrupados en cuatro categorías por simplicidad (identificadas como A-D, las cuales contienen 2, 5, 9 y 4 grafos de tareas con 2, 4, 5 y 6 tareas, respectivamente). Las columnas 3 y 4 corresponden al planificador software, mientras que las columnas 5 y 6 corresponden al planificador hardware. Las columnas 3 y 5 muestran los tiempos de ejecución obtenidos cuando los grafos de tareas se ejecutan en el planificador software y hardware, respectivamente. Por otro lado, las columnas 4 y 6 evalúan el impacto de las penalizaciones introducidas por los planificadores software y hardware, respectivamente. De esta manera, *Columna 4 = (Columna 3 – Columna 2)/Columna 2 * 100* y *Columna 6 = (Columna 5 – Columna 2)/Columna 2 * 100*.

Tal y como se muestra en la tabla, el planificador software introduce penalizaciones en tiempo de ejecución que oscilan del 1% al 28%. Para aplicaciones cuyo tiempo de ejecución es muy elevado (como HOUGH, por ejemplo), estos retardos no son muy significativos (apenas 1.15%). Sin embargo, cuando los tiempos de ejecución de las aplicaciones son reducidos (por ejemplo, el caso *Pocket GL (A)*) estos retardos sí pueden introducir una penalización significativa en el funcionamiento del sistema. Por tanto, en estos casos resulta muy interesante utilizar la implementación hardware, la cual es entre 2 y 3 órdenes de magnitud más eficiente que la implementación software equivalente, tal y como se muestra en la tabla B.1.

Tabla B.1. Retardos introducidos por ambas versiones del planificador desarrollado

Grafo de tareas	Tiempo de ejecución inicial (ms)	Planificador software		Planificador hardware	
		Tiempo de ejecución (ms)	Penalización (%)	Tiempo de ejecución (ms)	Penalización (%)
JPEG	79	80.07	1.35	79.022	0.03
PARALLEL-JPEG	54	55.44	2.67	54.023	0.04
MPEG-1	37	38.27	3.43	37.023	0.06
HOUGH	94	95.08	1.15	94.022	0.02
POCKET GL (A)	4.10	5.25	28.05	4.122	0.54
POCKET GL (B)	16.02	17.60	9.86	16.042	0.14
POCKET GL (C)	26.89	28.56	6.21	26.912	0.08
POCKET GL (D)	48.75	50.50	3.59	48.772	0.05

B.4.3. Evaluación de prestaciones

Por último, en esta tesis he evaluado también la calidad de las planificaciones obtenidas por el algoritmo de planificación propuesto. Para ello, he utilizado las aplicaciones que aparecen en la tabla B.1.

En primer lugar, los resultados obtenidos han demostrado que la técnica de prebúsqueda es una herramienta muy poderosa para eliminar gran parte de las latencias por reconfiguración que se introducirían si las tareas se cargasen en el preciso instante en el que se necesitan (estrategia *on demand*). En este último caso, estas reconfiguraciones producen una penalización del 37% con respecto a los tiempos de ejecución iniciales de las tareas. Sin embargo, estas penalizaciones se reducen al 10% sólo cuando se usa prebúsqueda (y no se reutiliza ninguna tarea).

Estos resultados pueden mejorar aún más si se explotan las posibilidades de reutilización de las tareas, ya que la reconfiguración de una tarea reutilizada no

genera ninguna penalización. En este sentido, la estrategia de reemplazo LF+C ha demostrado ser clave en la obtención de estos resultados. Para evaluar el impacto de utilizar LF+C con respecto a otras estrategias de reemplazo, como LRU o LFD, hemos diseñado dos experimentos:

- En un **primer experimento** se han ejecutado las tareas pertenecientes al conjunto {JPEG, MPEG-1, Parallel-JPEG, Hough} de forma recurrente siguiendo patrones fijos de dos tareas. (Por ejemplo: JPEG – MPEG-1 – JPEG – MPEG-1 – …). Hemos evaluado todos los posibles patrones que resultan al seleccionar dos tareas de este conjunto (el orden entre ellas importa) y para diferentes números de unidades reconfigurables: de 3 a 9.
- En un **segundo experimento** se han ejecutado las tareas pertenecientes al conjunto *Pocket GL* siguiendo secuencias de 500 tareas seleccionadas aleatoriamente. En este caso, también hay muchas posibilidades de reutilización, porque, a pesar de haber 20 grafos de tareas en este conjunto, entre todos ellos tienen tan sólo 10 nodos diferentes.

La figura B.11 muestra la comparativa entre las estrategias de reemplazo LF+C, LFD y LRU para el primer experimento. Por otro lado, la figura B.12 muestra la comparativa entre LF+C integrada en un planificador ASAP y en un planificador que puede saltar eventos en la reconfiguración de algunas tareas y, por tanto, flexibiliza las reconfiguraciones. Esta figura también muestra los resultados correspondientes al primer experimento. No obstante, los resultados equivalentes al segundo experimento muestran una tendencia similar en ambos casos. Es importante recordar también que LFD obtiene los resultados óptimos a nivel de reutilizaciones, pero sólo puede aplicarse a sistemas estáticos en los que se conocen todos los eventos que van a suceder en el futuro. Por otro lado, LF+C ha sido específicamente diseñada para ser utilizada en tiempo de

ejecución, y no necesita ninguna información acerca de los grafos de tareas que se ejecutarán en el futuro.

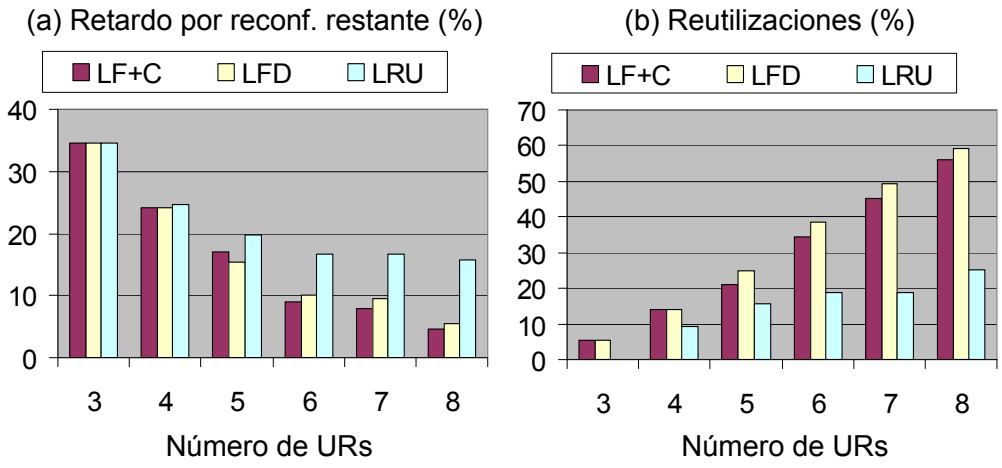

Figura B.11. Coste de implementación para el planificador hardware con diferentes números de unidades reconfigurables

En ambas figuras, el "*Retardo por reconf. Restante (%)*" muestra el porcentaje de penalizaciones por reconfiguración que se siguen produciendo a pesar de que las tareas se ejecutan con nuestro planificador. Por otro lado, el "*Reutilizaciones (%)*" muestra el porcentaje de tareas que se reutilizan con respecto al número de tareas que se ejecutan en total.

Tal y como muestra la figura B.11, LRU obtiene resultados pobres, ya que las penalizaciones que se siguen produciendo son aún muy altas (nunca menores del 15%) y las tasas de reutilización son muy bajas (apenas superan el 20% en el mejor de los casos). Sin embargo, LFD claramente mejora estos resultados y LF+C incluso mejora las prestaciones que se obtienen con LFD para 6, 7 y 8 unidades reconfigurables. Esto ocurre a pesar de que LFD conoce todas las tareas que se ejecutarán en el futuro, mientras que LF+C no dispone de esta información, lo cual hace que los resultados obtenidos sean aún más brillantes si cabe. Por otro lado, en la figura B.11.b también vemos que los porcentajes de

reutilización que se obtienen con LF+C son muy cercanos a los de LFD (los cuales conviene recordar que son los óptimos [Bela66]).

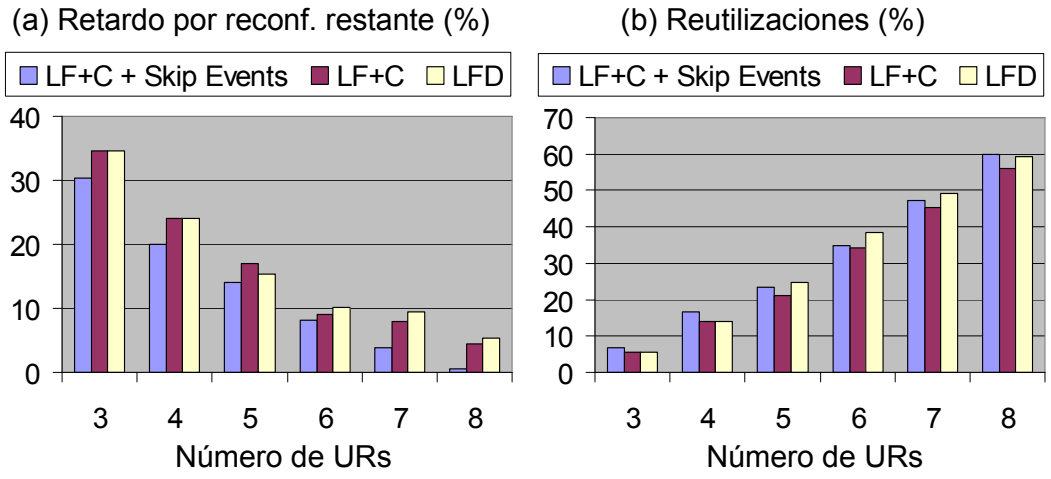

Figura B.12. Coste de implementación para el planificador hardware con diferentes números de unidades reconfigurables

Por otro lado, la figura B.12 muestra el impacto positivo de flexibilizar algunas reconfiguraciones (*LF+C + Skip Events*) en comparación con una estrategia de planificación puramente ASAP (*LF+C ASAP*). Tal y como se muestra en la figura, el retardo por reconfiguración restante disminuye de forma significativa con respecto a *LF+C ASAP* (de un 16.18% a un 12.89% de media). Por otro lado, los porcentajes de reutilización de *LF+C + Skip Events* también mejoran con respecto a *LF+C ASAP* (de un 29.30% a un 31.53% de media). Por último, si comparamos LFD y *LF+C + Skip Events*, vemos que ésta última obtiene en ocasiones mejores resultados que LFD en cuanto a penalizaciones ocultadas. Además, en algunos casos (para 3, 4 y 8 unidades reconfigurables), el porcentaje de reutilizaciones es mejor para *LF+C + Skip Events* con respecto a LFD; la cual, conviene recordar, es la estrategia que obtiene el porcentaje de reutilizaciones óptimo. Estos resultados pueden parecer extraños, ya que estamos consiguiendo "mejores resultados que los óptimos". Sin embargo, el motivo es que aplicando *LF+C + Skip Events*, nuestro planificador juega con

diferentes reglas, ya que LFD no puede retrasar ninguna reconfiguración, mientras que nuestro planificador aprovecha esta oportunidad de optimización para obtener resultados aún mejores.

B.5. Posibles líneas de trabajo futuro

Existen varias líneas de investigación interesantes para complementar este trabajo. Quizá la más directa sea ampliar este planificador para procesar varios grafos de tareas simultáneamente. Para ello, haría falta desarrollar técnicas de optimización que actúen fuera de los límites de un grafo de tareas. Otra extensión posible a nivel práctico para el trabajo de esta tesis sería integrar el planificador hardware en un sistema multitarea real basado en reconfiguración parcial. Ninguna de estas ampliaciones es trivial; es por ello por lo que me planteo estas optimizaciones a medio y largo plazo.

Otra posible línea de trabajo interesante a nivel teórico-práctico sería tener en cuenta el consumo energético de las reconfiguraciones durante el proceso de planificación. El módulo de reemplazo desarrollado en esta tesis ya reduce esta penalización al disminuir el número de reconfiguraciones. Sin embargo, si la penalización energética fuese muy elevada, sería interesante modificar el proceso de planificación para conseguir reducirlo aún más. De hecho, a nivel teórico estoy trabajando actualmente en adaptar la técnica de reemplazo propuesta en esta tesis a aumentar las tasas de reutilización de tareas en sistemas altamente dinámicos, en los que LFD no se puede aplicar directamente. A nivel práctico, podría ser interesante además poder evaluar con exactitud el consumo de potencia del circuito que realiza la reconfiguración. Para ello, habría dos posibilidades: en primer lugar, se puede modelar el circuito que realiza la reconfiguración y aplicar sobre él un conjunto de ecuaciones que modelen el consumo de energía, en segundo lugar podría realizarse una medición directa en un laboratorio.

Por último, otra posible extensión del planificador presentado en esta tesis sobre la que también estoy trabajando actualmente es trabajar con aplicaciones que no pueden ser representadas como DAGs (por ejemplo, porque tienen

pipelines). Esto es interesante ya que extenderá la aplicabilidad de las técnicas de planificación presentadas en esta tesis.

Generated publications

[CRM11] J. A. Clemente, J. Resano, and D. Mozos, "*A Replacement Technique to Maximize Task Reuse in Reconfigurable Systems*", in Reconfigurable Architectures Workshop (RAW), belonging to the IEEE International Parallel & Distributed Processing Symposium (IPDPS), (*Paper accepted, pending of publication*), 2011.

[CRGM10] J. A. Clemente, J. Resano, C. González and D. Mozos, "*A Hardware Implementation of a Run-Time Scheduler for Reconfigurable Systems*", in IEEE Transactions on Very Large Scale Integration Systems (TVLSI), vol. (*Pending of publication; available on-line*), pp. 1–14, 2010.

[CGRM10] J. A. Clemente, C. González, J. Resano and D. Mozos, "*A task graph execution manager for reconfigurable systems*", in Microprocessors and Microsystems: Embedded Hardware Design (MICPRO), vol. 34, pp. 73–83, 2010.

[RCG+08] J. Resano, J. A. Clemente, C. González, D. Mozos and F. Catthoor, "*Efficiently scheduling run-time reconfigurations*", in ACM Transactions on Design Automation of Electronic Systems (TODAES), vol.13, pp. 1–12, 2008.

[CGRM08a] J. A. Clemente, C. González, J. Resano and D. Mozos, "*A Hardware Task-Graph Scheduler for Reconfigurable Multi-tasking Systems*", in Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig), 2008, pp. 79–84.

[CGRM08b] J. A. Clemente, C. González, J. Resano and D. Mozos, "*Task-graph management for reconfigurable multi-tasking systems*", in Proceedings of the Reconfigurable Communication-centric Systems on Chip (ReCoSoc), 2008, pp. 192–193.

[CGRM08c] J. A. Clemente, C. González, J. Resano and D. Mozos, "*Implementaciones HW y SW de un gestor de ejecución de grafos de tareas en un sistema multitarea reconfigurable*", Actas de las Jornadas sobre Computación Reconfigurable y Aplicaciones (JCRA), 2008.

[RCG+07] J. Resano, J. A. Clemente, C. Gonzalez, J. L. García and D. Mozos, "*HW implementation of an execution manager for reconfigurable systems*", in Proceedings of the International Conference on Engineering Reconfigurable Systems and Algorithms (ERSA), 2007, pp. 71–77.

[GCG+07] C. González, J. A. Clemente, J. L. García, J. Resano and D. Mozos, "*Un sistema para la gestión eficiente del HW reconfigurable*", Actas de las Jornadas sobre Computación Reconfigurable y Aplicaciones (JCRA), 2007, pp. 163–170.

Bibliography

[AA09] A. Azarian and M. Ahmadi, "*Reconfigurable computing architecture survey and introduction*", in IEEE International Conference on Computer Science and Information Technology (ICCSIT), 2009, pp. 269-274.

[Bela66] L. A. Belady, "*A study of replacement algorithms for virtual storage computers*", in IBM Systems Journal, vol. 5, pp. 78–101, 1966.

[CRR+09] R. Cordone, F. Redaelli, M. A. Redaelli, M. D. Santambrogio and D. Sciuto, "*Partitioning and scheduling of Task Graphs on Partially Reconfigurable FPGAs*", in IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems (TCAD), vol. 28, pp. 662–675, 2009.

[CWB05] C. Chang, J. Wawrzynek and R. W. Brodersen, "*BEE2: A High-End Reconfigurable Computing System*", in IEEE Design & Test of Computers (D&T), vol. 22, pp. 114–125, 2005.

[DMB02] G. De Micheli and L. Benini, "*Networks on Chip: A New Paradigm for Systems on Chip Design*", in Proceedings of Design, Automation and Test in Europe (DATE), 2002, pp. 418–419.

[DMP06] K. D. Danne, R. Miihlenbernd and M. Platzner, "*Executing hardware tasks on dynamically reconfigurable devices under real-time conditions*", in Proceedings of the International Conference on Field Programmable Logic and Applications (FPL), 2006, pp. 1–6.

[DP05] A. Dandalis and V. K. Prasanna, "*Configuration Compression for FPGA-based Embedded Systems*", in IEEE Transactions on Very Large Scale Integration Systems (TVLSI), vol. 13, pp. 1394–1398, 2005.

[DR02] – J. Daemen and V. Rijmen, "*The Design of Rijndael: AES - The Advanced Encryption Standard*", Springer-Verlag, 2002.

[ECF96] C. Ebeling, D. Cronquist and P. Franklin, "*RaPiD - reconfigurable pipelined datapath*", in Proceedings of the International Workshop on Field Programmable Logic and Applications (FPL), 1996, pp. 126–135.

[EIA85] Electronics Industries Association, "*EIA Standard RS-232-C Interface Between Data Terminal Equipment and Data Communication Equipment Employing Serial Data Interchange*", August 1969, reprinted in Telebyte Technology Data Communication Library, Greenlawn, NY, 1985.

[EKO90] H. Eschenauer, J. Koski and A. Osyczka, "*Multicriteria Design Optimization*", Springer-Verlag, 1990.

[Estr60] G. Estrin, "*Organization of Computer Systems - The Fixed Plus Variable Structure Computer*", in Proceedings of the Western Joint Computer Conference, 1960, pp. 33–40.

[FFM+99] T. Fujii, K. Furuta, M. Motomura, M. Nomura, M. Mizuno, K. Anjo, K. K. Wakabayashi, Y. Hirota, Y. Nakazawa, H. Ito and M. Yamashina, "*A dynamically reconfigurable logic engine with a multi-context/multi-mode unified-cell architecture*", in Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC), 1999, pp. 364–365.

[GC08] H. Gu and S. Chen, "*Partial Reconfiguration Bitstream Compression for Virtex FPGAs*", in Proceedings of the Congress on Image and Signal Processing (CISP), 2008, pp. 183–185.

[GNS04] S. Ghiasi, A. Nahapetian, and M. Sarrafzadeh, "*An Optimal Algorithm for Minimizing Run-Time Reconfiguration Delay*", in ACM Transactions on Embedded Computing Systems (TECS), vol. 3, pp. 237–256, 2004.

[Harte01] R. Hartenstein, "*Coarse grain reconfigurable architectures*", in Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC), 2001, pp. 564–570.

[Hauc98] S. Hauck, "*Configuration Prefetch for Single Context Reconfigurable Coprocessors*", in Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), 1998, pp. 65–74.

[HLS98] S. Hauck, Z. Li and E. Schwabe, "*Configuration compression of the Xilinx XC6200 FPGA*", in Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 1998, pp. 138–146.

[HM04] J. Hu and R. Marculescu, "*Energy-aware communication and task scheduling for network-on-chip architectures under real-time constraints*", in Proceedings of Design, Automation and Test in Europe (DATE), 2004, pp. 234–239.

[HP95] J. L. Hennessy and D. A. Patterson, "*Computer architecture: a quantitative approach*", second edition, Morgan Kaufmann, 1995, pp. 760.

[Huff52] D. A. Huffman, "*A Method for the Construction of Minimum-Redundancy Codes*", in Proceedings of the Institute of Radio Engineers (IRE), 1952, pp. 1098–1101.

[HW90] F. Harris and D. Wright, "*The JPEG Algorithm for Image Compression: A Software Implementation and some Test Results*", in Conference Record Twenty-Fourth Asilomar Conference on Signals, Systems and Computers (ACSSC), 1990, pp. 870.

[KMK07] K. Kosciuszkiewicz, F. Morgan and K. Kepa, "*Run-time management of reconfigurable hardware tasks using embedded Linux*", in Proceedings of the International Conference on Field Programmable Technology (ICFPT), 2007, pp. 209–215.

[KSS+03] J.-K. Kim, S. Shivle, H. J. Siegel, A. A. Maciejewski, T. D. Braun, M. Schneider, S. Tideman, R. Chitta, R. B. Dilmaghani, R. Joshi, A. Kaul, A. Sharma, S. Sripada, P. Vangari and S. S. Yellampalli, "*Dynamic mapping in a*

heterogeneous environment with tasks having priorities and multiple deadlines", in Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS), 2003, pp. 98–113.

[LH01] Z. Li and S. Hauck, "*Configuration Compression for Virtex FPGAs*", in Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2001, pp. 147–159.

[LH02] Z. Li and S. Hauck, "*Configuration Prefetching Techniques for Partial Reconfigurable Coprocessors with Relocation and Defragmentation*", in Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), 2002, pp. 187–195.

[Li00] Z. Li, "*Configuration Cache Management Techniques for Reconfigurable Computing*", in Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2000, pp. 22–36.

[LMM91] X. Lai, J. L. Massey and S. Murphy, "*Markov ciphers and differential cryptanalysis*", in Advances in Cryptology (CRYPTO), Springer-Verlag, 1991, pp. 17–38.

[Mare07] T. Marescaux, "*Introducing the SuperGT Network-on-Chip: SuperGT QoS: more than just GT*", in Proceedings of the Design Automation Conference (DAC), 2007, pp. 116–121.

[Mark92] B. D. Markey, "*HyTime and MHEG*" in Proceedings of the 37th IEEE International Conference, 1992, pp. 25–40.

[MBC07] T. Marescaux, E. Brockmeyer and H. Corporaal, "*The impact of higher communication layers on NOC supported MP-SoCs*", in Proceedings of the International Symposium on Networks-on-Chip (NOCS), 2007, pp. 107–116.

[MBV+02] T. Marescaux, A. Bartic, D. Verkest, S. Vernalde, and R. Lauwereins, "*Interconnection networks enable fine-grain dynamic multi-tasking on FPGAs*", in Proceedings of the International Conference on Field Programmable Logic and Applications (FPL), 2002, pp. 795–805.

[MDH96] E. Mirsky and A. DeHon, "*MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources*", in Proceedings of the IEEE Symposium on FPGAs for custom computing machines (FCCM), 1996, pp. 157–166.

[MMB+03] T. Marescaux, J-Y. Mignolet, A. Bartic, W. Moffat, D. Verkest and S. Vernalde, "*Networks on Chip as Hardware Components of an OS for Reconfigurable Systems*", in Proceedings of the International Conference on Field Programmable Logic and Applications (FPL), 2003, pp. 595–605.

[MRS+07] S. Mellers, B. Richards, H. K.-H. So, S. M. Mishra, K. Camera, P. A. Subrahmanyam and R. W. Brodersen, "*Radio Testbeds Using BEE2*", in Conference Record of the Asilomar Conference on Signals, Systems and Computers (ACSSC), 2007, pp. 1991–1995.

[NB01] J. Noguera and R. M. Badía, "*A HW/SW Partitioning Algorithm for Dynamically Reconfigurable Architectures*", in Proceedings of Design, Automation and Test in Europe (DATE), 2001, pp. 729–734.

[NB02a] J. Noguera and R. M. Badía, "*HW/SW Codesign Techniques for Dynamically Reconfigurable Architectures*", in Transactions on Very Large Scale Integration Systems (TVLSI), vol. 10, pp. 399–415, 2002.

[NB02b] J. Noguera and R. M. Badía, "*Dynamic Run-Time HW/SW Scheduling Techniques for Reconfigurable Architectures*", in Proceedings of the International Symposium on Hardware/Software Codesign (CODES), 2002, pp. 205–210.

[NB04] J. Noguera and R. M. Badía, "*Multitasking on Reconfigurable Architectures: Microarchitecture Support and Dynamic Scheduling*", in ACM Transactions on Embedded Computing Systems (TECS), vol. 3, pp. 385–406, 2004.

[NMA+05] V. Nollet, T. Marescaux, P. Avasare, D. Verkest and J.-Y. Mignolet, "*Centralized run-time management in a network on chip containing reconfigurable hardware tiles*", in Proceedings of Design, Automation and Test in Europe (DATE), 2005, pp. 234–239.

[NoII97] P. Noll; "*MPEG digital audio coding*", in IEEE Signal Processing Magazine, vol.14, pp.59–81, 1997.

[PRMC07] E. Pérez, J. Resano, D. Mozos and F. Catthoor, "*Reducing the reconfiguration overhead: a survey of techniques*", in Proceedings of the International Conference on Engineering Reconfigurable Systems and Algorithms (ERSA), 2007, pp. 191–194.

[QSN06] Y. Qu, J.-P. Soininen and J. Nurmi, "*A Parallel Configuration Model for Reducing the Run-Time Reconfiguration Overhead*", in Proceedings of Design, Automation and Test in Europe (DATE), 2006, pp. 1–6.

[Riss76] J. J. Rissanen, "*Generalized Kraft Inequality and Arithmetic Coding*", in IBM Journal of Research and Development, vol. 20, pp. 198–203, 1976.

[RMC05] J. Resano, D. Mozos and F. Catthoor, "*A Hybrid Prefetch Scheduling Heuristic to Minimize at Run-Time the Reconfiguration Overhead of Dynamically Reconfigurable Hardware*", in Proceedings of Design, Automation and Test in Europe (DATE), 2005, pp. 106–111.

[RMVC05] J. Resano, D. Mozos, D. Verkest and F. Catthoor, "*A Reconfiguration Manager for Dynamically Reconfigurable Hardware*", in IEEE Design & Test of Computers (D&T), vol. 22, pp. 452–460, 2005.

[RVM+04] J. Resano, D. Verkest, D. Mozos, S. Vernalde and F. Catthoor, "*A hybrid design-time/run-time scheduling flow to minimize the reconfiguration overhead of FPGAs*", in Microprocessors and Microsystems: Embedded Hardware Design (MICPRO), vol. 28, pp. 291–301, 2004.

[SB08a] H. K.-H. So and R. W. Brodersen, "*Runtime File System Support for Reconfigurable FPGA Hardware Processes in BORPH*", in Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2008, pp. 285–286.

[SB08b] H. K.-H. So and R. W. Brodersen, "*File system access from reconfigurable FPGA hardware processes in BORPH*", in Proceedings of the

International Conference on Field Programmable Logic and Applications (FPL), 2008, pp. 567–570.

[SHWK09] B. Sellers, J. Heiner, M. Wirthlin and J. Kalb, "*Bitstream compression through frame removal and partial reconfiguration*", in Proceedings of the International Conference on Field Programmable Logic and Applications (FPL), 2009, pp. 476–480.

[SLL+00] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh and E. M. Chaves Filho, "*Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications*", in IEEE Transactions on Computers (TC), vol. 49, pp 465–481, 2000.

[SNG01] S. Sudhir, S. Nath and S. Goldstein, "*Configuration caching and swapping*", in Proceedings of the International Conference on Field Programmable Logic and Applications (FPL), 2001, pp. 27–29.

[SRM09] M. D. Santambrogio, M. Redaelli and M. Maggioni, "*Task Graph Scheduling for Reconfigurable Architectures driven by Reconfigurations Hiding and Resources Reuse*", in Proceedings of the ACM Great Lakes Symposium on VLSI (GLSVLSI), 2009, pp. 21–26.

[STB06] H. K.-H. So, A. Tkachenko and R. W. Brodersen, "*A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH*", in Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2006, pp. 259–264.

[SUA+00] Y. Shibata, M. Uno, H. Amano, K. Furuta, T. Fujii and M. Motomura, "*A virtual hardware system on a dynamically reconfigurable logic devices*", in Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2000, pp. 295–296.

[TCB06] A. Tkachenko, D. Cabric and R. W. Brodersen, "*Cognitive Radio Experiments using reconfigurable BEE2*", in Conference Record of the Asilomar

Conference on Signals, Systems and Computers (ACSSC), 2006, pp. 2041–2045.

[TCW+05] T. J. Todman, G. A. Constantinides, S. J. E. Wilton, O. Mencer, W. Luk and P. Y. K. Cheung, "*Reconfigurable computing: architectures and design methods*", in IET Computers & Digital Techniques, vol. 152, pp. 193–207, 2005.

[WK02] G. B. Wigley and D. A. Kearney, "*Research Issues in Operating Systems for Reconfigurable Computing*", in Proceedings of the International Conference on Engineering of Reconfigurable System and Algorithms (ERSA), 2002, pp. 10–16.

[WMY01] C. Wong, P. Marchal and P. Yang, "*Task concurrency management methodology to schedule the MPEG4 IM1 player on a highly parallel processor platform*", in Proceedings of the International Symposium on Hardware/Software Codesign (CODES), 2001, pp. 170–175.

[WP03] H. Walder and M. Platzner, "*Online Scheduling for Block-Partitioned Reconfigurable Devices*", in Proceedings of Design, Automation and Test in Europe (DATE), 2003, pp. 290–295.

[WP04] H. Walder and M. Platzner, "*A Runtime Environment for Reconfigurable Hardware Operating Systems*", in Proceedings of the Field Programmable Logic and Applications (FPL), 2004, pp. 831–835.

[YC03] P. Yang and F. Catthoor, "*Pareto-Optimization-Based Run-Time Task Scheduling for Embedded Systems*", in Proceedings of the First IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (ISSS), 2003, pp. 120–125.

[YC04] P. Yang and F. Catthoor, "*Dynamic mapping and ordering tasks of embedded real-time systems n multiprocessor platforms*", in Proceedings of the International Workshop on Software and Compilers for Embedded Systems (SCOPES), 2004; Lecture Notes in Computer Science 3199, pp. 167–181.

[YWM+01] P. Yang, C. Wong, P. Marchal, F. Catthoor, D. Desmet, D. Verkest and R. Lauwereins, "*Energy-Aware Runtime Scheduling for Embedded-*

Multiprocessors SOCs", in IEEE Design & Test of Computers (D&T), pp. 46–58, 2001.

[ZL77] J. Ziv and A. Lempel, "*A universal algorithm for data compression*", in IEEE Transactions on Information Theory (TIT), vol. 23, pp. 337–343, 1977.